12 Interacting with servers using HTTP

This chapter covers

- Working with the HttpClient service
- Creating a simple web server using the Node and Express frameworks
- Developing an Angular client that communicates with the Node server
- Intercepting HTTP requests and responses

Angular applications can communicate with any web server supporting HTTP, regardless of what server-side platform is used. In this chapter, we'll show you how to use the `HttpClient` service offered by Angular. You'll see how to make HTTP GET and POST methods with `HttpClient`. And you'll learn how to intercept HTTP requests to implement cross-cutting concerns, such as global error handling.

This chapter starts with a brief overview of Angular's `HttpClient` service, and then you'll create a web server using the Node.js and Express.js frameworks. The server will serve the data required for most code samples in this chapter.

Finally, you'll see how to implement HTTP interceptors and report progress while transferring large assets.

12.1 Overview of the HttpClient service

Browser-based web apps run HTTP requests asynchronously, so the UI remains responsive, and the user can continue working with the application while HTTP requests are being processed by the server. Asynchronous HTTP requests can be implemented using callbacks, promises, or observables. Although promises allow you to move away from callback hell (see section A.12.2 in appendix A), they have the following shortcomings:

- There's no way to cancel a pending request made with a promise.
- When a promise resolves or rejects, the client receives either data or an error message, but in both cases it'll be a single piece of data. A JavaScript promise doesn't offer a way to handle a continuous stream of data chunks delivered over time.

Observables don't have these shortcomings. In section 6.4 in chapter 6, we demonstrated how you can cancel HTTP requests made with observables, and in chapter 13, you'll see how a server can push a stream of data to the client using WebSockets.

Angular supports HTTP communications via the `HttpClient` service from the `@angular/common/http` package. If your app requires HTTP communications, you need to add `HttpClientModule` to the `imports` section of the `@NgModule()` decorator.

If you peek inside the type definition file @angular/common/http/src/client.d.ts, you'll see that `get()`, `post()`, `put()`, `delete()`, and many other methods return an `Observable`, and an app needs to subscribe to get the data. To use the `HttpClient` service, you need to inject it into a service or component.

> **NOTE** As explained in chapter 5, every injectable service requires a provider declaration. The providers for `HttpClient` are declared in `HttpClientModule`, so you don't need to declare them in your app.

The following listing illustrates one way of invoking the `get()` method of the `HttpClient` service, passing a URL as a `string`. You retrieve products of type `Product` here.

Listing 12.1 Making an HTTP GET request

```
interface Product {      <-- Defines the type Product
     id: number,
    title: string
}
...
constructor(private httpClient: HttpClient) { }  <-- Injects the HttpClient service

ngOnInit() {
  this.httpClient.get<Product>('/product/123')  <-- Declares a get() request
       .subscribe(
```

```
      data => console.log(`id: ${data.id} title: ${data.title}`),
       (err: HttpErrorResponse) => console.log(`Got error: ${err}`)
    );
}
```

Subscribes to the result of get()

Logs an error, if any

In the `get()` method, you haven't specified the full URL (such as http://localhost:8000/product) assuming that the Angular app makes a request to the same server where it was deployed, so the base portion of the URL can be omitted. Note that in `get<Product>()`, you use TypeScript generics (see section B.9 in appendix B) to specify the type of data expected in the body of the HTTP response. The type annotation doesn't enforce or validate the shape of the data returned by the server; it just makes the other code aware of the expected server response. By default, the response type is `any`, and the TypeScript compiler won't be able to type-check the properties you access on the returned object.

Your `subscribe()` method receives and prints the data on the browser's console. By default, `HttpClient` expects the data in JSON format, and the data is automatically converted into JavaScript objects. If you expect non-JSON data, use the `responseType` option. For example, you can read arbitrary text from a file as shown in the following listing.

Listing 12.2 Specifying string as a returned data type

```
let someData: string;

this.httpClient
    .get<string>('/my_data_file.txt', {responseType: 'text'})
     .subscribe(
        data => someData = data,
       (err: HttpErrorResponse) => console.log(`Got error: ${err}`)
     );
```

Specifies string as a response body type

Logs errors, if any

Assigns the received data to a variable

TIP The `post()`, `put()`, and `delete()` methods are used in a fashion similar to listing 12.2 by invoking one of these methods and subscribing to the results.

Now let's create an app that reads some data from a JSON file.

12.2 Reading a JSON file with HttpClient

To illustrate `HttpClient.get()`, your app will read a file containing JSON-formatted product data. Create a new folder that contains the products.json file shown in the following listing.

Listing 12.3 The file data/products.json

```
[
  { "id": 0, "title": "First Product", "price": 24.99 },
  { "id": 1, "title": "Second Product", "price": 64.99 },
  { "id": 2, "title": "Third Product", "price": 74.99}
]
```

The folder data and the products.json file become assets of your app that need to be included in the project bundles, so you'll add this folder to the app's `assets` property in the .angular-cli.json file (or angular.json, starting from Angular 6), as shown in the next listing.

Listing 12.4 A fragment from .angular-cli.json

```
"assets": [
  "assets",
   "data"
 ],
```

The name of the default assets folder generated by Angular CLI

The name of the folder with assets you add to the project

Let's create an app that will show the product data, as shown in figure 12.1.

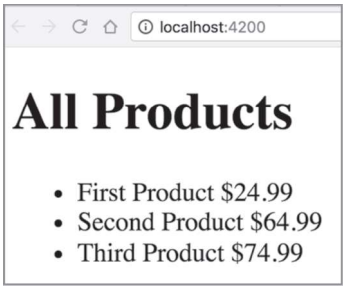


Figure 12.1 Rendering the content of products.json

Your app component will use `HttpClient.get()` to issue an HTTP GET request, and you'll declare a `Product` interface defining the structure of the expected product data. The observable returned by `get()` will be unwrapped in the template by the `async` pipe. The app.component.ts file is located in the readfile folder and has the content shown in the following listing.

Listing 12.5 app.component.ts

```
interface Product {
   id: string;
  title: string;
  price: number;
}

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

Declares a product type

Iterates through the observable products and autosubscribes to them with the async pipe

Renders the product title and the price formatted as currency

```
export class AppComponent{

  products$: Observable<Product[]>;

  constructor(private httpClient: HttpClient) {
     this.products$ = this.httpClient
                         .get<Product[]>('/data/products.json');
   }
}
```

Declares a typed observable for products

Injects the HttpClient service

Makes an HTTP GET request specifying the type of the expected data

NOTE In this app, you don't use the lifecycle hook `ngOnInit()` for fetching data. That's not a crime, because this code doesn't use any component properties that may not have been initialized during component construction. This data fetch will be executed asynchronously after the constructor when the `async` pipe subscribes to the `products$` observable.

To see this app in action, run `npm install` in the client directory, and then run the following command:

```
ng serve --app readfile -o
```

It wasn't too difficult, was it? Open Chrome Dev tools, and you'll see the HTTP request and response and their headers, as shown in figure 12.2.

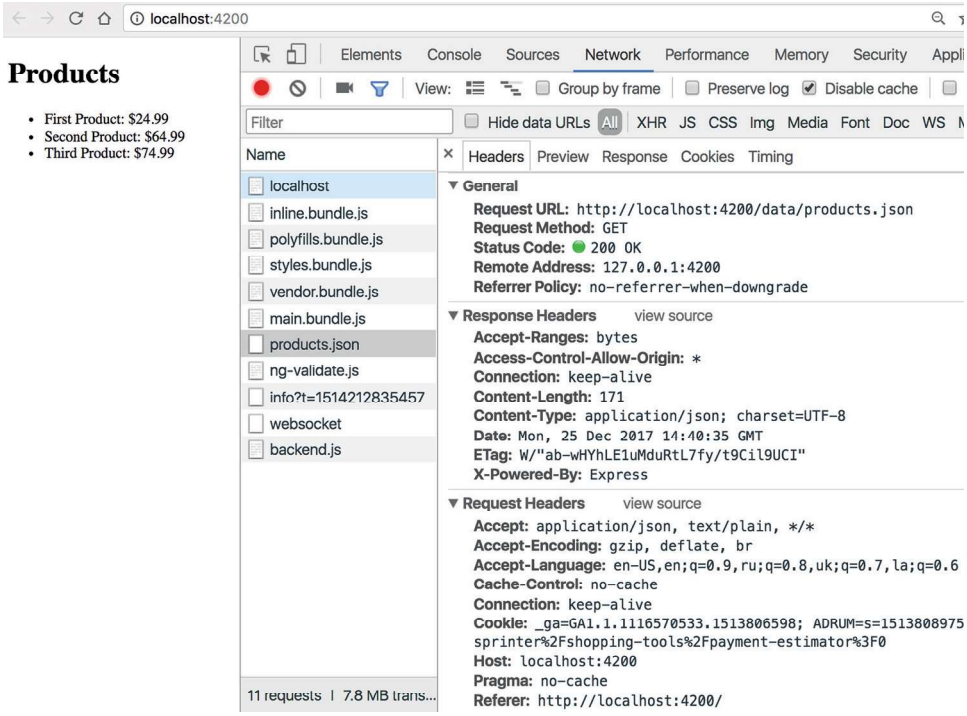


Figure 12.2 Monitoring HTTP request and response

This app illustrates how to make an HTTP GET request that has no parameters and uses default HTTP request headers. If you want to add additional headers and query parameters, use an overloaded version of the `get()` method that offers an extra parameter where you can specify additional options. The following listing shows how to request data from the same products.json file, passing additional headers and query parameters, using the classes `HttpHeaders` and `HttpParams`.

Listing 12.6 Adding HTTP headers and query parameters to the GET request

```
constructor(private httpClient: HttpClient) {

    let httpHeaders = new HttpHeaders()
       .set('Content-Type', 'application/json')
       .set('Authorization', 'Basic QWxhZGRpb');

    let httpParams = new HttpParams()
       .set('title', "First");

    this.products$ = this.httpClient.get<Product[]>('/data/products.json',
      {
       headers: httpHeaders,
      params: httpParams
    });
```

Creates the HttpHeaders object with two additional headers

Creates the object with one query parameter (it can be any object literal)

Passes the headers and query parameters as a second argument of get()

Since you simply read a file, passing query parameters doesn't make much sense, but if you needed to make a similar request to a server's endpoint that knows how to search products by title, the code would look the same. Using the chainable `set()` method, you can add as many headers or query parameters as needed.

Running listing 12.7 renders the same data from products.json, but the URL of the request and HTTP headers will look different. Figure 12.3 uses arrows to highlight the differences compared to figure 12.2.

You may be wondering how to send data (for example, using HTTP POST) to the server. To write such an app, you need a server that can accept your data. In section 12.4.2, you'll create an app that uses `HttpClient.post()`, but first let's create a web server using the Node.js and Express.js frameworks.

12.3 Creating a web server with Node, Express, and TypeScript

Angular can communicate with web servers running on any platform, but we decided to create and use a Node.js server in this book for the following reasons:

- There's no need to learn a new programming language to understand the code.
- Node allows you to create standalone applications (such as servers).
- Using Node lets you continue writing code in TypeScript, so we don't have to explain how to create a web server in Java, .NET, or any other language.

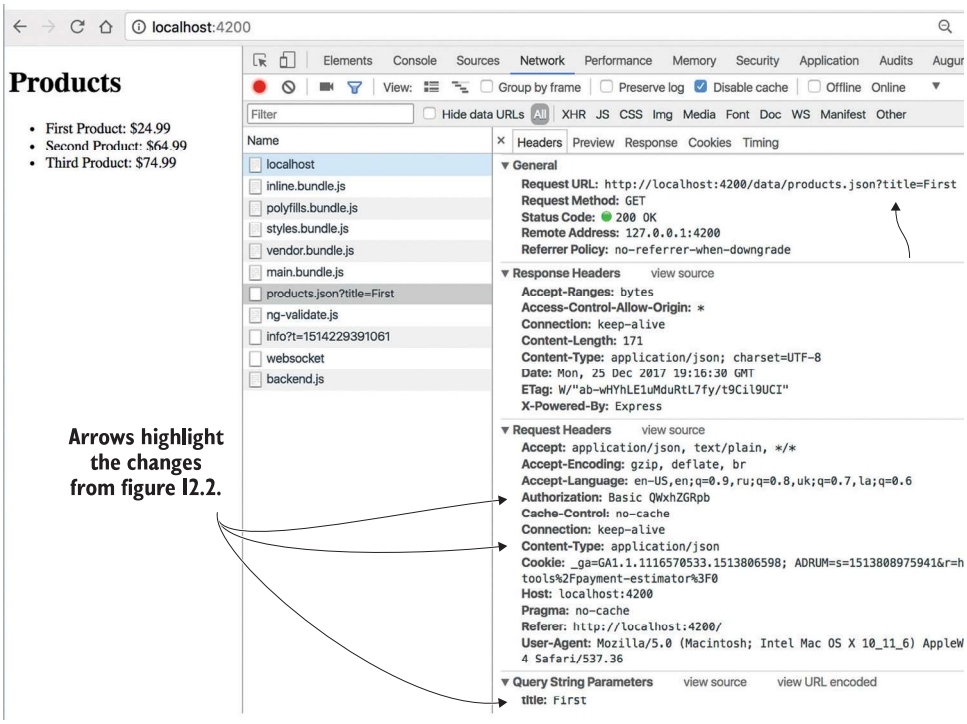


Figure 12.3 Monitoring the modified HTTP request

You'll start with writing a basic web server using Node and Express frameworks. Then, you'll write another web server that can serve JSON data using the HTTP protocol. After this server's ready, you'll create an Angular client that can consume its data.

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

12.3.1 Creating a simple web server

In this section, you'll create a web server using Node and Express (http://expressjs.com) frameworks and TypeScript. The code that comes with this chapter has a directory called server, containing a separate project with its own package .json file, which doesn't include any Angular dependencies. The sections for `dependencies` and `devDependencies` of this file look like the following listing.

Listing 12.7 The server's dependencies in package.json

```
"dependencies": {
    "express": "^4.16.2",   <-- Express.js framework
     "body-parser": "^1.18.2"   <-- Request body parser for Express.js
  },
```

```
"devDependencies": {
  "@types/express": "^4.0.39",
   "@types/node": "^6.0.57",
   "typescript": "^2.6.2"
 }
```

Type definition files for Express.js

Type definition files for Node.js

Local version of the TypeScript compiler

You can read about type definition files in section B.12 in appendix B. You'll use the body-parser package for extracting the data from a request object in section 12.4.

> **NOTE** You install the local version of the TypeScript compiler just in case you need to keep a different version of the `tsc` compiler installed globally. Also, you shouldn't expect that a continuous integration server has a global `tsc` executable. To use the local `tsc` version, you can add a custom npm script command in the `scripts` section of package.json (`"tsc": "tsc"`) and start the compiler by running the `npm run tsc` command .

Because you'll write the server code in TypeScript, it needs to be transpiled, so the following listing adds the tsconfig.json file with the compiler's options for `tsc`.

Listing 12.8 tsconfig.json for the server

```
{
  "compilerOptions": {
    "module": "commonjs",
     "outDir": "build",
     "target": "es6"
   },
  "exclude": [
    "node_modules"
   ]
}
```

Transpiles modules according to the CommonJS spec

Saves .js files into the build directory

Transpiles into .js files using ES6 syntax

Doesn't transpile the code located in the node_modules directory

By specifying the CommonJS syntax for modules, you ensure that `tsc` transpiles statements like `import * as express from "express";` into `const express = require ("express");`, as required by Node.

The following listing shows the code of a simple web server from the file my-express-server.ts. This server implements the server-side routing for HTTP `GET` requests by mapping three paths—/, /products, and /reviews—to the corresponding callback functions.

Listing 12.9 my-express-server.ts

```
import * as express from "express";
const app = express();

app.get('/', (req, res) => res.send('Hello from Express'));

app.get('/products', (req, res) => res.send('Got a request for products'));
```

Instantiates Express.js

Matches GET requests to the root route

Matches GET requests to the /products route

```
app.get('/reviews', (req, res) => res.send('Got a request for reviews'));
const server = app.listen(8000, "localhost", () => {
   console.log(`Listening on localhost:8000`);
});
```

Matches GET requests to the /reviews route

Starts listening on localhost:8000 and executes the code from the fat-arrow function

Run `npm install`; transpile all code samples, including my-express-server.ts, by running `tsc`; and start this server by running the following command:

```
node build/my-express-server
```

> **NOTE** If you don't have the TypeScript compiler installed globally, you can either run its local version, ./node_modules/typescript/bin/tsc, or add the line `"tsc: "tsc"` to the `scripts` section of package.json, and run it like this: `npm run tsc`.

You'll see the message "Listening on localhost:8000" on the console, and now you can request either products or reviews, depending on which URL you enter in the browser, as shown in figure 12.4.

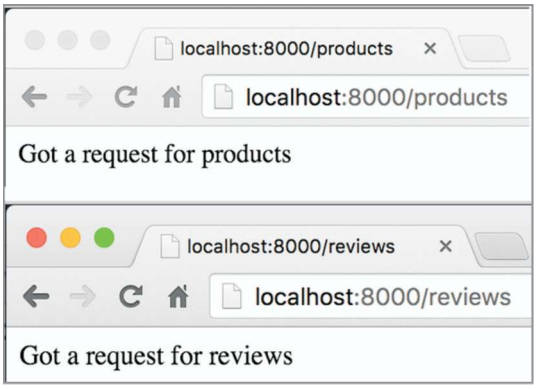


Figure 12.4 Server-side routing with Express

> **NOTE** To debug Node applications, refer to the documentation of your IDE. If you want to debug the TypeScript code, don't forget to set the option `"sourceMap": true` in the tsconfig.json file of your Node project.

This server responds with simple text messages, but how do you create a server that can respond with data in JSON format?

12.3.2 Serving JSON

To send JavaScript objects (such as products) to the browser in JSON format, you'll use the Express function `json()` on the response object. Your REST server is located in the rest-server.ts file, and it can serve either all products or a specific one (by ID). In this

server, you'll create three endpoints: / for the root path, /api/products for all products, and /api/products/:id for the paths that include product IDs. The `products` array will contain three hardcoded objects of type `Product`, which will be turned into JSON format by invoking `res.json()`, offered by the Express framework.

Listing 12.10 rest-server.ts

```
import * as express from "express";
const app = express();

interface Product {
     id: number,
    title: string,
    price: number
}

const products: Product[] = [
    {id: 0, title: "First Product", price: 24.99},
    {id: 1, title: "Second Product", price: 64.99},
    {id: 2, title: "Third Product", price: 74.99}
];

function getProducts(): Product[] {
     return products;
}

app.get('/', (req, res) => {
     res.send('The URL for products is http://localhost:8000/api/products');
});

app.get('/api/products', (req, res) => {
     res.json(getProducts());
});

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

app.get('/api/products/:id', (req, res) => {
     res.json(getProductById(parseInt(req.params.id)));
 });

const server = app.listen(8000, "localhost", () => {
    console.log(`Listening on localhost:8000`);
});
```

Defines the Product type

Creates an array of three JavaScript objects with products data

Returns the text prompt as a response to the base URL GET request

When the GET request contains /api/products in the URL, invokes getProducts()

Returns all products

Converts products to JSON and returns them to the browser

Returns the product by ID. Here, you use the array's find() method.

Converts the product ID from a string to an integer, invokes getProductById(), and sends the JSON back

The GET request came with a parameter. Its values are stored in the params property of the request object.

Stop the my-express-server from the previous section if it's running (Ctrl-C), and start the rest-server with the following command:

```
node build/rest-server
```

Enter `http://localhost:8000/api/products` in the browser, and you should see the data in JSON format, as shown in figure 12.5.

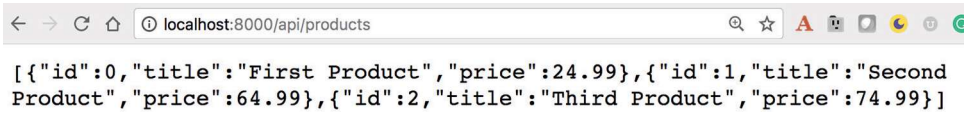


Figure 12.5 The server's response to http://localhost:8000/api/products

Figure 12.6 shows the browser window after you enter the URL http://localhost:8000/api/products/1. This time, the server returns only data about the product that has an `id` with the value of `1`.

Figure 12.6 The server's response to http://localhost:8000/api/products/1

Your REST server is ready. Now let's see how to initiate HTTP GET requests and handle responses in Angular applications.

12.4 Bringing Angular and Node together

In the preceding section, you created the rest-server.ts file, which responds to HTTP GET requests with product details regardless of whether the client was written using a framework or the user simply entered the URL in a browser. In this section, you'll write an Angular client that will issue HTTP GET requests and treat the product data as an `Observable` data stream returned by your server.

> **NOTE** Just a reminder: the Angular app and the Node server are two separate projects. The server code is located in the directory called server, and the Angular app is located in a separate project in the client directory.

12.4.1 Static assets on the server

A typical web app deployed on the server includes static assets (for example, HTML, images, CSS, and JavaScript code) that have to be loaded by the browser when the user enters the URL of the app. From the server's perspective, the Angular portion of a web app is considered *static assets*. The Express framework allows you to specify the directory where the static assets are located.

Let's create a new server: rest-server-angular.ts. In the rest-server.ts file from the previous section, you didn't specify the directory with static assets, because no client

app was deployed on the server. In the new server, you add the lines shown in the following listing.

Listing 12.11 Specifying the directory with static resources

```
import * as path from "path";

app.use('/', express.static(path.join(__dirname, 'public')));
```

Adds the Node path module for working with the directory and paths

Assigns the public subdirectory as the location of the static resources

Unlike in rest-server.ts, you just map the base URL (/) to the public directory, and Node will send index.html from there by default. The browser loads index.html, which in turn loads the rest of the bundles defined in the `<script>` tags.

> **NOTE** The original index.html file generated by Angular CLI doesn't contain `<script>` tags, but when you run the `ng build` or `ng serve` commands, they create a new version of index.html that includes the `<script>` tags with all the bundles and other assets.

When the browser requests static assets, Node will look for them in the public subdirectory of the current one (`__dirname`)—the build directory from which you started this server. Here, you use Node's `path.join()` API to ensure that the absolute file path is created in a cross-platform way. In the next section, we'll introduce the Angular client and deploy its bundles in the public directory. The REST endpoints in rest-server-angular.ts remain the same as in rest-server.ts:

- / serves index.html, which contains the code to load the Angular app.
- /api/products serves all products.
- /api/products/:id serves one product by its ID.

The complete code of the rest-server-angular.ts file is shown in the next listing.

Listing 12.12 rest-server-angular.ts

```
import * as express from "express";
import * as path from "path";

const app = express();

app.use('/', express.static(path.join(__dirname, 'public')));

interface Product {
    id: number,
    title: string,
    price: number
}

const products: Product[] = [
    {id: 0, title: "First Product", price: 24.99},
```

Adds the path module that provides utilities for working with file and directory paths

For the root path, specifies the directory from which to serve static assets

```
    {id: 1, title: "Second Product", price: 64.99},
    {id: 2, title: "Third Product", price: 74.99}
];

function getProducts(): Product[] {
    return products;
}

app.get('/api/products', (req, res) => {
     res.json(getProducts());
});

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

app.get('/api/products/:id', (req, res) => {
     res.json(getProductById(parseInt(req.params.id)));
});

const server = app.listen(8000, "localhost", () => {
     console.log(`Listening on localhost:8000`);
});
```

Configures the endpoint for HTTP GET requests

Configures another endpoint for HTTP GET requests

Starts the server

The new server is ready to serve JSON data to the Angular client, so let's start it:

```
node build/rest-server-angular
```

Trying to make a request to this server using the base URL http:// localhost:8000 will return a 404 error, because the directory with static assets doesn't contain the index.html file: you haven't deployed your Angular app there yet. Your next task is to create and deploy the Angular app that will consume JSON-formatted data.

12.4.2 Consuming JSON in Angular apps

Your Angular app will be located in a directory called client. In previous chapters, you were starting all Angular apps building bundles in memory with `ng serve`, but this time you'll also use the `ng build` command to generate bundles in files. Then you'll use npm scripts to automate the deployment of these bundles in the Node server created in section 12.4.1.

In dev mode, you'll keep serving Angular apps using the dev server from Angular CLI that runs on port 4200. But the data will be coming from another web server, powered by Node and Express, that will run on port 8000. Figure 12.7 illustrates this two-server setup.

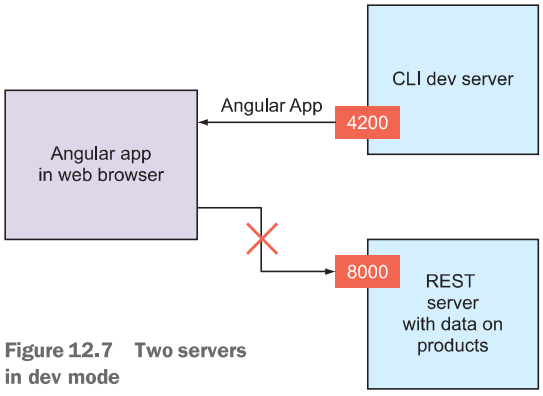


Figure 12.7 Two servers in dev mode

NOTE *Spoiler alert*: We'll run into an issue when the client app served from one server tries to directly access another one. We'll cross that bridge when we get to it.

When the Angular `HttpClient` object makes a request to a URL, the response comes back as `Observable`, and the client's code can handle it either by using the `subscribe()` method or with the `async` pipe introduced in section 6.5 in chapter 6. Using the `async` pipe is preferable, but we'll show you both methods so you can appreciate the advantages of `async`.

Let's start with an app that retrieves all products from the rest-server-angular server and renders them using an HTML unordered list (`<ul>`). You can find this app in the app.component.ts file located in the client/src/app/restclient directory.

Listing 12.13 restclient/app.component.ts

```
// import statements omitted for brevity
interface Product {
   id: number,
  title: string,
  price: number
}

@Component({
  selector: 'app-root',
  template: `<h1>All Products</h1>
  <ul>
    <li *ngFor="let product of products">
       {{product.title}}: {{product.price | currency}}
     </li>
  </ul>
  {{error}}
  `})
export class AppComponent implements OnInit {

  products: Product[] = [];
  theDataSource$: Observable<Product[]>;
   productSubscription: Subscription;
   error: string;

  constructor(private httpClient: HttpClient) {
     this.theDataSource$ = this.httpClient
              .get<Product[]>('http://localhost:8000/api/products');
   }
  ngOnInit() {
    this.productSubscription = this.theDataSource$
     .subscribe(
        data => this.products = data,
        (err: HttpErrorResponse) =>
         this.error = `Can't get products. Got ${err.message}`
      );
  }
}
```

Declares the type for products

Uses the currency pipe for rendering the price

Declares an observable for data returned by HttpClient

Declares the subscription property—you'll need to unsubscribe from observable

The HTTP requests errors (if any) are displayed here.

Injects HttpClient

Declares the intention to issue HTTP GET for products

Makes an HTTP GET request for products

Sets the value of an error message to a variable for rendering on the UI

Adds the received products to the array

NOTE You didn't use `ngOnDestroy()` to explicitly unsubscribe from the observable because once `HttpClient` gets the response (or an error), the underlying `Observable` completes, so the observer is unsubscribed automatically.

You already started the server in the previous section. Now, start the client by running the following command:

```
ng serve --app restclient -o
```

No products are rendered by the browser, and the console shows a 404 error, but if you used the full URL in the `AppComponent` (for example, http://localhost:8000/api/products), the browser's console would show the following error:

```
Failed to load http://localhost:8000/api/products:
  ➥No 'Access-Control-Allow-
     Origin' header is present on the requested resource.
  ➥Origin 'http://localhost:4200' is therefore not allowed access.
```

That's because you violated the same-origin policy (see http://mng.bz/2tSb). This restriction is set for clients that run in a browser as a security mechanism. Say you visited and logged in to bank.com, and then opened another tab and opened badguys.com. The same-origin policy ensures that scripts from badguys.com can't access your account at bank.com.

Your Angular app was loaded from http://localhost:4200 but tries to access the URL http://localhost:8000. Browsers aren't allowed to do this unless the server that runs on port 8000 is configured to allow access to the clients with the origin http://localhost:4200. When your client app is deployed in the Node server, you won't have this error, because the client app will be loaded from the server that runs on port 8000, and this client will be making data requests to the same server.

In the hands-on section in chapter 13, you'll use the Node.js CORS package (see https://github.com/expressjs/cors) to allow requests from clients with other origins, but this may not be an option if you need to make requests to third-party servers. In dev mode, there's a simpler solution to the same-origin restriction. You'll use the server that runs on port 4200 as a proxy for client requests to the server that runs on port 8000. The same-origin policy doesn't apply to server-to-server communications. In the next section, you'll see how to configure such a proxy on the client.

12.4.3 Configuring the client proxy

In dev mode, you'd like to continue using the server that comes with Angular CLI with its hot reload features and fast rebuilding of application bundles in memory. On the other hand, you want to be able to make requests to other servers.

Under the hood, the Angular CLI dev server uses the Webpack dev server, which can serve as a proxy mediating browser communications with other servers. You just need to create a proxy-conf.json file in the root directory of the Angular project, where you'd configure a URL fragment(s) that the dev server should redirect to

another server. In your case, you want to redirect any request that has the URL fragment /api to the server that runs on port 8000, as shown in the following listing.

Listing 12.14 proxy-conf.json

```
{
  "/api": {
     "target": "http://localhost:8000",
     "secure": false
   }
}
```

Hijacks all requests that have /api in the URL

Redirects these requests to this URL

The target connection doesn't need SSL certificates.

NOTE Using a proxy file allows you to easily switch between local and remote servers. Just change the value of the `target` property to have your local app retrieve data from a remote server. You can read more about Angular CLI proxying support at http://mng.bz/fLgf.

You need to make a small change in the app from the preceding section. You should replace the full URL of the backend server (http://localhost:8000/api/products) with the path of the endpoint (/api/products). The code that makes a request for products will look as if you try to access the /api/products endpoint on the Angular CLI dev server where the app was downloaded from:

```
this.theDataSource = this.httpClient
              .get<Product[]>('/api/products');
```

But the dev server will recognize the /api fragment in the URL and will redirect this request to another server that runs on port 8000, as shown in figure 12.8 (compare it to figure 12.7).

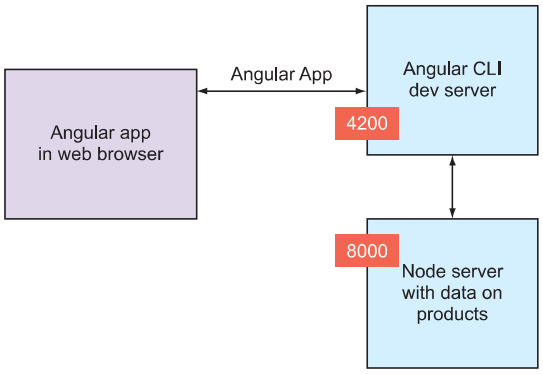


Figure 12.8 Two servers with a proxy

To see the modified app in action, you need to use the `--proxy-config` option, providing the name of the file where you configured the proxy parameters:

```
ng serve --app restclient --proxy-config proxy-conf.json -o
```

NOTE If you forget to provide the name of the proxy file when configuring proxy parameters, you'll get a 404 error, because the /api/products request won't be redirected, and there's no such endpoint in the server that runs on port 4200.

Open your browser to http://localhost:4200, and you'll see the Angular app shown in figure 12.9.

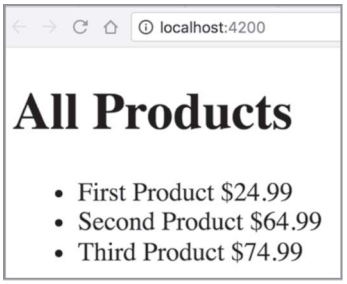


Figure 12.9 Retrieving all products from the Node server via proxy

Note that data arrives from the server that runs on port 4200, which got it from the server that runs on port 8000. Figure 12.8 illustrates this data flow.

In dev mode, using Angular CLI proxying allows you to kill two birds with one stone: have the hot reload of your app on any code change and access data from another server without the need to deploy the app there.

Now let's see how to replace the explicit subscription for products with the `async` pipe.

12.4.4 Subscribing to observables with the async pipe

We introduced `AsyncPipe` (or `async`, when used in templates) in section 6.5 of chapter 6. `async` can receive an `Observable` as input, autosubscribe to it, and discard the subscription when the component gets destroyed. To see this in action, make the following changes in listing 12.13:

- Change the type of the `products` variable from `Array` to `Observable`.
- Remove the declaration of the variable `theDataSource$`.
- Remove the invocation of `subscribe()` in the code. You'll assign the `Observable` returned by the `get()` method to `products`.
- Add the `async` pipe to the `*ngFor` loop in the template.

The following listing implements these changes (see the file restclient/app.component.asyncpipe.ts).

Listing 12.15 app.component.asyncpipe.ts

```
import { HttpClient} from '@angular/common/http';
import {Observable, EMPTY} from 'rxjs';
import {catchError} from 'rxjs/operators';
import {Component} from "@angular/core";

interface Product {
  id: number,
  title: string,
  price: number
}

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

```
  <ul>
    <li *ngFor="let product of products$ | async">
       {{product.title }} {{product.price | currency}}
    </li>
  </ul>
  {{error}}
  `})
export class AppComponentAsync{

  products$: Observable<Product[]>;
  error: string;
  constructor(private httpClient: HttpClient) {
    this.products$ = this.httpClient.get<Product[]>('/api/products')
       .pipe(
        catchError( err => {
           this.error = `Can't get products. Got ${err.status} from ${err.url}`;
           return EMPTY;
          });
  }
}
```

The async pipe subscribes and unwraps products from observable products$.

Initializes the observable with HttpClient.get()

Handles the error, if any

Returns an empty observable so the subscriber won't get destroyed

Intercepts an error before it reaches the async pipe

Running this application will produce the same output shown in figure 12.9.

So far, you've been injecting `HttpClient` instances directly into components, but more often you inject `HttpClient` into a service. Let's see how to do this.

12.4.5 Injecting HttpClient into a service

Angular offers an easy way for separating the business logic implementation from rendering the UI. Business logic should be implemented in services, and the UI in components, and you usually implement all HTTP communications in one or more services that are injected into components. For example, your ngAuction app that comes with chapter 11 has the `ProductService` class with the injected `HttpClient` service. You inject a service into another service.

`ProductService` reads the products.json file using `HttpClient`, but it could get the product data from a remote server the same way you did in the previous section. `ProductService` is injected into components of ngAuction. Check the source code of `ProductService` and `CategoriesComponent` in ngAuction that comes with chapter 11, and you'll recognize the pattern shown in figure 12.10.

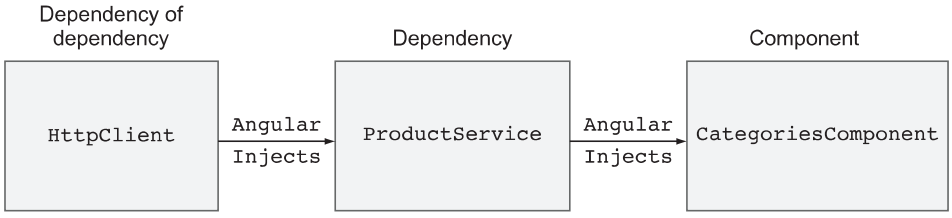


Figure 12.10 Injecting into a service and a component

The following listing from ngAuction's `ProductService` is an example of encapsulating business logic and HTTP communications inside a service.

Listing 12.16 A fragment from ngAuction's `ProductService`

```
@Injectable()
export class ProductService {
  constructor(private http: HttpClient) {}

  getAll(): Observable<Product[]> {
    return this.http.get<Product[]>('/data/products.json');
   }

  getById(productId: number): Observable<Product> {
    return this.http.get<Product[]>('/data/products.json')
       .pipe(
        map(products => products.find(p => p.id === productId))
      );
  }

  getByCategory(category: string): Observable<Product[]> {
    return this.http.get<Product[]>('/data/products.json').pipe(
       map(products => products.filter(p => p.categories.includes(category)))
     );
  }

  getDistinctCategories(): Observable<string[]> {
   return this.http.get<Product[]>('/data/products.json')
     .pipe(
      map(this.reduceCategories),
      map(categories => Array.from(new Set(categories))),
    );
  }
  // Other code is omitted for brevity
}
```

Injects HttpClient into ProductService

Invokes HttpClient.get()

NOTE In the preceding listing 12.19, you use RxJS pipeable operators inside the `pipe()` method (see section D.4.1 in appendix D).

The next listing from `CategoriesComponent` is an example of using the preceding service in the component.

Listing 12.17 A fragment from the ngAuction's `CategoriesComponent`

```
@Component({
  selector: 'nga-categories',
  styleUrls: [ './categories.component.scss' ],
  templateUrl: './categories.component.html'
})
export class CategoriesComponent {
  readonly categoriesNames$: Observable<string[]>;
  readonly products$: Observable<Product[]>;

  constructor(
    private productService: ProductService,
```

Injects ProductService

```
    private route: ActivatedRoute
  ) {
    this.categoriesNames$ = this.productService.getDistinctCategories()
        .pipe(map(categories => ['all', ...categories]));

    this.products$ = this.route.params.pipe(
      switchMap(({ category }) => this.getCategory(category)));
  }

  private getCategory(category: string): Observable<Product[]> {
    return category.toLowerCase() === 'all'
      ? this.productService.getAll()
      : this.productService.getByCategory(category.toLowerCase());
  }
}
```

Uses ProductService

The provider for `ProductService` is declared on the app level in the `@NgModule()` decorator of the root module of ngAuction. In the hands-on section in chapter 13, you'll split ngAuction into two projects, client and server, and the web server will be written using Node and Express frameworks. How can an Angular app (bundles and assets) be deployed in a web server?

12.4.6 Deploying Angular apps on the server with npm scripts

The process of deploying the code of a web client in a server should be automated. At the very minimum, deploying an Angular app includes running several commands for building bundles and replacing previously deployed code (index.html, JavaScript bundles, and other assets) with new code. The deployment may also include running test scripts and other steps.

JavaScript developers use various tools for automating running deployment tasks such as Grunt, gulp, npm scripts, and others. In this section we'll show you how to use npm scripts for deployment. We like using npm scripts, because they're simple to use and offer an easy way to automate running command sequences in a predefined order. Besides, you already have npm installed, so there's no need to install additional software for automating your deployment workflow.

To illustrate the deployment process, you'll use the rest-server-angular server from section 12.3.1, where you'll deploy the Angular app from section 12.3.4. After deployment, you won't need to configure the proxy anymore, because both the server and the client code will be deployed at the same server running http:// localhost:8000. After entering this URL in the browser, the user will see the product data, as shown earlier in figure 12.9.

npm allows you to add the `scripts` property in package.json, where you can define aliases for terminal commands. For example, instead of typing the long command `ng serve --app restclient --proxy-config proxy-conf.json`, you can define a start command in the scripts section of package.json as follows:

```
"scripts": {
  "start": "ng serve --app restclient --proxy-config proxy-conf.json"
}
```

Now, instead of typing that long command, you'll just enter `npm start` in the console. npm supports more than a dozen script commands right out of the box (see the npm-scripts documentation for details, https://docs.npmjs.com/misc/scripts). You can also add new custom commands specific to your development and deployment workflow.

Some of these scripts need to be run manually (such as `npm start`), and some are invoked automatically if they have the `post` and `pre` prefixes (for example, `post-install`). If any command in the `scripts` section starts with the `post` prefix, it'll run automatically after the corresponding command specified after this prefix.

For example, if you define the command `"postinstall": "myCustomInstall.js"`, each time you run `npm install`, the script myCustomInstall.js will automatically run right after. Similarly, if a command has a `pre` prefix, such a command will run before the command named after this prefix.

If you define custom commands that aren't known by npm scripts, you'll need to use an additional option: `run`. Say you defined a custom command `startDev` like this:

```
"scripts": {
  "startDev": "ng serve --app restclient --proxy-config proxy-conf.json"
}
```

To run that command, you need to enter the following in your terminal window: `npm run startDev`. To automate running some of your custom commands, use the same prefixes: `post` and `pre`.

Let's see how to create a sequence of runnable commands for deploying an Angular app on the Node server. Open package.json from the client directory, and you'll find four custom commands there: `build`, `postbuild`, `predeploy`, and `deploy`. The following listing shows what will happen if you run a single command: `npm run build`.

Listing 12.18 A fragment from client/package.json

The command ng build will create a production build of the restclient app in the default directory dist.

Since there's a postbuild command, it starts automatically and will try to run the deploy command.

```
"scripts": {
  "build": "ng build --prod --app restclient",
  "postbuild": "npm run deploy",
  "predeploy": "rimraf ../server/build/public && mkdirp ../server/build/public",
  "deploy": "copyfiles -f dist/** ../server/build/public"
}
```

Finally, the deploy command is executed.

Since there's also a predeploy command there, it'll run after the postbuild and before deploy.

We'll explain what the commands `predeploy` and `deploy` do in a minute, but our main message here is that starting a single command resulted in running four commands in the specified order. Creating a sequence of deployment commands is easy.

> **TIP** If you build the bundles with AOT compilation and use only standard Angular decorators (no custom ones), you can further optimize the size of the JavaScript in your app by commenting out the line `import 'core-js/es7/reflect';` in the polyfills.ts file. This will reduce the size of the generated polyfill bundle.

Typically, the deployment process removes the directory with previously deployed files, creates a new empty directory, and copies the new files into this directory. In your deployment scripts, you use three npm packages that know how to do these operations, regardless of the platform you use (Windows, Unix, or macOS):

- `rimraf`—Removes the specified directory and its subdirectories
- `mkdirp`—Creates a new directory
- `copyfiles`—Copies files from source to destination

Check the devDependencies section in package.json, and you'll see `rimraf`, `mkdirp`, and `copyfiles` there.

> **TIP** Currently, Angular CLI uses Webpack to build bundles. Angular CLI 7 will come with new build tools. In particular, it'll include Closure Compiler, which produces smaller bundles.

The code that comes with this chapter is located in two sibling directories: client and server. Your `predeploy` command removes the content of the server/build/public directory (this is where you'll deploy the Angular app) and then creates a new empty public directory. The `&&` sign allows you to define commands that run more than one script.

The `deploy` command copies the content of the client/dist directory (the app's bundles and assets) into server/build/public.

In the real world, you may need to deploy an Angular app on a remote server, so using the package copyfiles won't work. Consider using an SCP utility (see https://en.wikipedia.org/wiki/Secure_copy) that performs secure file transfer from a local computer to a remote one.

If you can manually run a utility from the terminal window, you can run it using npm scripts as well. In chapter 14, you'll learn how to write test scripts. Including a test runner into your build process could be as simple as adding `&& ng test` to your `predeploy` command. If you find some useful gulp plugins, create the npm script for it, for example, `"myGulpCommand" : "gulp SomeUsefulTask"`.

To see that your deployment scripts work, perform the following steps:

1. Start the server by running the following command in the server directory:

   ```
   node build/rest-server-angular
   ```

2. In the client directory, run the build and deployment scripts:

   ```
   npm run build
   ```

   Check the server/build/public directory—the client's bundles should be there.

3 Open your browser to http://localhost:8000, and your Angular app will be loaded from your Node server showing three products, as shown in figure 12.9.

We've described the entire process of creating and running a web server, as well as creating and running Angular apps in dev mode and deploying in the server. Your Angular app was using the `HttpClient` service to issue HTTP GET requests to retrieve data from the server. Now let's see how to issue HTTP POST requests to post data to the server.

12.5 Posting data to the server

HTTP POST requests are used for sending new data to the server. With `HttpClient`, making POST requests is similar to making GET requests. Invoking the `HttpClient.post()` method declares your intention to post data to the specified URL, but the request is made when you invoke `subscribe()`.

NOTE For updating existing data on the server, use `HttpClient.put()`; and for deleting data, use `HttpClient.delete()`.

12.5.1 Creating a server for handling post requests

You need a web server with an endpoint that knows how to handle POST requests issued by the client. The code that comes with this chapter includes a server with an /api/product endpoint for adding new products, located in the rest-server-angular-post.ts file. Because your goal isn't to have a fully functional server for adding and saving products, the /api/product endpoint will simply log the posted data on the console and send a confirmation message to the client.

The posted data will arrive in the request body, and you need to be able to parse it to extract the data. The npm package body-parser knows how to do this in Express servers. If you open package.json in the server directory, you'll find body-parser in the dependencies section. The entire code of your server is shown in the following listing.

Listing 12.19 rest-server-angular-post.ts

```
import * as express from "express";
import * as path from "path";
import * as bodyParser from "body-parser";

const app = express();

app.use('/', express.static(path.join(__dirname, 'public')));

app.use(bodyParser.json());

app.post("/api/product", (req, res) => {

  console.log(`Received new product
              ${req.body.title} ${req.body.price}`);
```

Adds the body-parser package

Creates the parser to turn the payload of req.body into JSON

Creates an endpoint for handling POST requests

Logs the payload of the POST request

```
  res.json({'message':`Server responded: added ${req.body.title}`});
 });

const server = app.listen(8000, "localhost", () => {
  const {address, port} = server.address();
  console.log(`Listening on ${address}: ${port}`);
});
```

Sends the confirmation message to the client

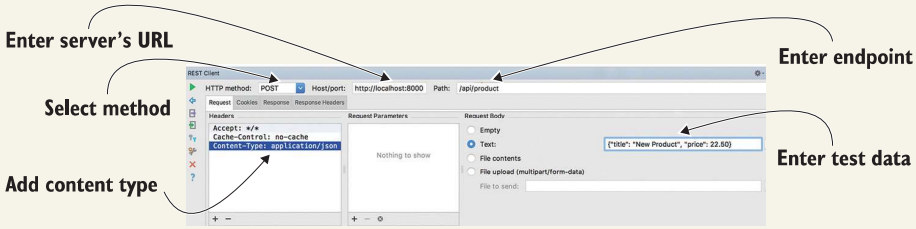
Your server expects the payload in a JSON format, and it'll send the response back as a JSON object with one property: `message`. Start this server by running the following command in the server directory (don't forget to run `tsc` to compile it):

```
node build/rest-server-angular-post
```

Testing a RESTful API

When you create web servers with REST endpoints, you should test them to ensure that the endpoints work properly even before you start writing any client code. Your IDE may offer such a tool. For example, the WebStorm IDE has a menu item, Test RESTful Web Service, under Tools. After entering all the data to test your server, this tool looks like the following figure.

Using WebStorm's Test RESTful client



The server response

Press the green play button, and you'll see the response of your /api/product endpoint under the Response tab, as shown in the next figure.

The server response

If your IDE doesn't offer such a testing tool, use the Chrome extension called Advanced REST Client (https://install.advancedrestclient.com/#/install) or a tool called Postman (www.getpostman.com).

Now that you've created, started, and tested the web server, let's write the Angular client that will post new products to this server.

12.5.2 Creating a client for making post requests

Your Angular app will render a simple form where the user can enter the product title and price, as shown in figure 12.11.

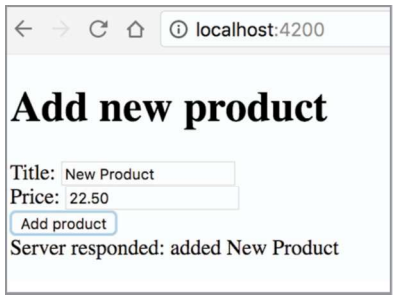


Figure 12.11 UI for adding new products

After filling out the form and clicking the Add Product button, the server will respond with the confirmation message shown under the button.

In this app, you'll use the template-driven Forms API, and your form will require the user to enter the new product's title and price. On the button click, you'll invoke the method `HttpClient.post()`, followed by `subscribe()`.

The code of this Angular app is located in the client directory under the restclient-post subdirectory. It's shown in the following listing.

Listing 12.20 app.component.ts

```
import {Component} from "@angular/core";
import {HttpClient, HttpErrorResponse} from "@angular/common/http";

@Component({
  selector: 'app-root',
  template: `<h1>Add new product</h1>
     <form #f="ngForm" (ngSubmit) = "addProduct(f.value)" >
       Title: <input id="productTitle" name="title" ngModel>
```

```
        <br>
        Price: <input id="productPrice" name="price" ngModel>
        <br>
        <button type="submit">Add product</button>
      </form>
      {{response}}
  `})
export class AppComponent {

  response: string;

  constructor(private httpClient: HttpClient) {}

  addProduct(formValue) {

    this.response='';

    this.httpClient.post<string>("/api/product",
                                 formValue)
      .subscribe(
        data =>  this.response = data['message'],
        (err: HttpErrorResponse) =>
            this.response = `Can't add product. Error code:
             ${err.message} ${err.error.message}`
      );
  }
}
```

Declares the intention to make a POST request

Makes the HTTP POST request

Provides the POST payload

Gets the server's response

Handles errors

When the user clicks Add Product, the app makes a POST request and subscribes to the server response. `formValue` contains a JavaScript object with the data entered in the form, and `HttpClient` automatically turns it into a JSON object. If the data was posted successfully, the server returns a JSON object with the `message` property, which is rendered using data binding.

If the server responds with an error, you display it in the UI. Note that you use `err.message` and `err.error.message` to extract the error description. The second property may contain additional error details. Modify the code of the server to return a string instead of JSON, and the UI will show a detailed error message.

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

```
ng serve --app restclientpost --proxy-config proxy-conf.json -o
```

You now know how to send HTTP requests and handle responses, and there could be lots of them in your app. Is there a way to intercept all of them to provide some additional processing, like showing/hiding the progress bar, or logging requests?

12.6 HTTP interceptors

Angular allows you to create HTTP interceptors for pre- and post-processing of all HTTP requests and responses of your app. They can be useful for implementing such cross-cutting concerns as logging, global error handling, authentication, and others. We'd like to stress that the interceptors work *before* the request goes out or before a response is rendered on the UI. This gives you a chance to implement the fallback scenarios for certain errors or prevent attempts of unauthorized access.

To create an interceptor, you need to write a service that implements the `HttpInterceptor` interface, which requires you to implement one method: `intercept()`. Angular will provide two arguments to this callback: `HttpRequest` and `HttpHandler`. The first one contains the request object being intercepted, which you can clone and modify. The second argument is used to forward the modified request to the backend or another interceptor in the chain (if any) by invoking the `handle()` method.

NOTE The `HttpRequest` and `HttpResponse` objects are immutable, and the word *modify* means creating and passing through the new instances of these objects.

The interceptor service shown in the following listing doesn't modify the outgoing `HttpRequest` but simply prints its content on the console and passes it through as is.

Listing 12.21 A simple interceptor

```
@Injectable()
export class MyFirstInterceptor implements HttpInterceptor {

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    // Clone and modify your HTTPRequest using req.clone()
    // or perform other actions here

    console.log("I've intercepted your HTTP request! ${JSON.stringify(req)}`);

    return next.handle(req);
  }
}
```

In listing 12.21, you forward the `HttpRequest`, but you could modify its headers or parameters and return the modified request. The `next.handle()` method returns an observable when the request is complete, and if you want to modify the HTTP response as well, apply additional RxJS operators on the stream returned by `next.handle()`.

The `intercept()` method receives the `HttpRequest` object and returns, not the `HttpResponse` object, but the observable of `HttpEvent`, because Angular implements `HttpResponse` as a stream of `HttpEvent` values.

Both `HttpRequest` and `HttpResponse` are immutable, and if you want to modify their properties, you need to clone them first, as in the following listing.

Listing 12.22 Modifying `HTTPRequest`

```
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any
     >> {
    const modifiedRequest = req.clone({
      setHeaders: { ('Authorization', 'Basic QWxhZGRpb') }
    });
    return next.handle(modifiedRequest);
  }
```

Because an interceptor is an injectable service, don't forget to declare its provider for the `HTTP_INTERCEPTORS` token in the `@NgModule()` decorator:

```
providers: [{provide: HTTP_INTERCEPTORS,
             useClass: MyFirstInterceptor, multi: true}]
```

The `multi: true` option tells you that `HTTP_INTERCEPTORS` is a multiprovider token—an array of services can represent the same token. You can register more than one interceptor, and Angular will inject all of them:

```
providers: [{provide: HTTP_INTERCEPTORS,
               useClass: MyFirstInterceptor, multi: true},
            {provide: HTTP_INTERCEPTORS,
               useClass: MySecondInterceptor, multi: true}]
```

NOTE If you have more than one interceptor, they'll be invoked in the order they're defined.

To illustrate how interceptors work, let's create an app with an `HttpInterceptor` that will intercept and log all errors returned by the server. For the client, you'll reuse the app from section 12.5.2 shown in figure 12.11, adding the logging service and the interceptor to log errors on the console.

You'll slightly modify the server from the previous section to randomly generate errors. You can find the complete code of the server in the rest-server-angular-post-errors.ts file. Now, instead of just responding with success messages, it'll randomly return an error, as shown in the following listing.

Listing 12.23 Emulating server errors

```
if (Math.random() < 0.5) {
     res.status(500);
    res.send({'message': `Server responded: error adding product
                          ${req.body.title}`});
} else {
     res.send({'message': `Server responded: added ${req.body.title}`);
}
```

Returns an HTTP response with the status 500

Returns a successful HTTP response

Start this server as follows:

```
node build/rest-server-angular-post-errors
```

Your Angular app is located in the interceptor directory and includes a logging service implemented as two classes: `LoggingService` and `ConsoleLoggingService`. `LoggingService` is an abstract class that declares one method, `log()`.

Listing 12.24 logging.service.ts

```
@Injectable()
export abstract class LoggingService {

  abstract log(message: string): void;
}
```

Because this class is abstract, it can't be instantiated, and you'll create the class `ConsoleLoggingService` shown in the following listing.

Listing 12.25 console.logging.service.ts

```
@Injectable()
export class ConsoleLoggingService implements LoggingService{

   log(message:string): void {
        console.log(message);
   }
}
```

You may be wondering why you create the abstract class for such a simple logging service. It's because in real-world apps, you may want to introduce logging, not only on the browser's console, but also on the server. Having an abstract class would allow you to use it as a token for declaring a provider:

```
providers: [{provide: LoggingService, useClass: ConsoleLoggingService}]
```

Later on, you can create a class called `ServerLoggingService` that implements `LoggingService`, and to switch from console to server logging, you'll need to change the provider without having to modify components that use it:

```
providers: [{provide: LoggingService, useClass: ServerLoggingService}]
```

If your interceptor receives an error, you'll do the following:

1. Log it on the console.
2. Replace the `HttpErrorResponse` with a new instance of `HttpResponse` that will contain the error message.
3. Return the new `HttpResponse` so the client can show it to the user.

The interceptor class will use the `catchError` operator on the observable returned by `HttpHandler.next()`, where you'll implement these steps. Your interceptor is implemented in the logging.interceptor.service.ts file.

Listing 12.26 logging.interceptor.service.ts

```
import {Injectable} from "@angular/core";
import {HttpErrorResponse, HttpEvent, HttpHandler,
      HttpInterceptor, HttpRequest, HttpResponse} from "@angular/common/http";
import {Observable, of} from "rxjs";
```

```
import {catchError} from 'rxjs/operators';
import {LoggingService} from "./logging.service";

@Injectable()
export class LoggingInterceptor implements HttpInterceptor {

  constructor(private loggingService: LoggingService) {}

  intercept(req: HttpRequest<any>, next: HttpHandler):
➥Observable<HttpEvent<any>> {
    return next.handle(req)
        .pipe(
          catchError((err: HttpErrorResponse) =>
            this.loggingService.log(`Logging Interceptor: ${err.error.message}`);
            return of(new HttpResponse(
                {body:{message: err.error.message}}));
          })
      );
  }
}
```

Injects the console logging service

Forwards requests to the server and responses to the client

The new HttpResponse will contain the error message.

Replaces HttpErrorResponse with HttpResponse

Logs the error message on the console

Catches the response errors returned by the server

The code for the application component has no references to the interceptor class, as you'll see in the following listing. It'll be always receiving `HttpResponse` objects that contain either a message that the server successfully added a new product, or an error message.

Listing 12.27 app.component.ts

```
import {Component} from "@angular/core";
import {HttpClient} from "@angular/common/http";
import {Observable} from "rxjs";
import {map} from "rxjs/operators";

@Component({
  selector: 'app-root',
  template: `<h1>Add new product</h1>
  <form #f="ngForm" (ngSubmit) = "addProduct(f.value)" >
    Title: <input id="productTitle" name="title" ngModel>
    <br>
    Price: <input id="productPrice" name="price" ngModel>
    <br>
    <button type="submit">Add product</button>
  </form>
  {{response$ | async}}
   `})
export class AppComponent {

  response$: Observable<string>;

  constructor(private httpClient: HttpClient) {}
```

Renders any messages received from the server (including errors)

This observable is for the interceptor's responses.

```
  addProduct(formValue){
    this.response$=this.httpClient.post<{message: string}>("/api/product",
       formValue)
      .pipe(
        map (data=> data.message)
      );
  }
}
```

Expects the server's responses to HTTP POST as {message: string}

Extracts the text of the message property

When you compare this app with the one from the previous section, note that you don't handle errors in the component, which renders the messages to the UI. Now the `LoggingInterceptor` will handle all HTTP errors.

To see this app in action, run the following command and monitor the browser console for logging messages.

```
ng serve --app interceptor --proxy-config proxy-conf.json -o
```

This app should give you an idea of how to implement a cross-cutting concern like a global error-logging service for all HTTP responses without the need to modify any application components or services that use the `HttpClient` service.

An HTTP request runs asynchronously and can generate a number of progress events that you might want to intercept and handle. Let's look at how you'd do that.

12.7 Progress events

Sometimes uploading or downloading certain assets (like large data files or images) takes time, and you should keep the user informed about the progress. `HttpClient` offers progress events that contain information like total size of the asset, current number of bytes that are already uploaded or downloaded, and more.

To enable progress events tracking, make your requests using the `HttpRequest` object with the option `{reportProgress: true}`. For example, you can make an HTTP GET request that reads the my_large_file.json file.

Listing 12.28 Making a GET request with events tracking

```
const req = new HttpRequest('GET',
                            './my_large_file.json',
                            { reportProgress: true });
 httpClient.request(req).subscribe(
        // Handle progress events here);
```

Declares an intention to make a GET request

Specifies the file to read

Enables progress event

Makes a request

In the `subscribe()` method , check whether the emitted value is an event of the type you're interested in, for example, `HttpEventType.DownloadProgress` or `HttpEventType.UploadProgress`. These events have the `loaded` property for the current number of transferred bytes and the `total` property, which knows the total size of the transfer.

The next app shows how to handle a progress event for calculating and showing the percentage of the file download. This app comes with a large 48 MB JSON file. The content of the file is irrelevant in this case. Figure 12.12 shows the app when the download of the file is complete. The percentage on the left is changing as this file is being loaded by `HttpClient`. This app also reports the progress on the browser's console.

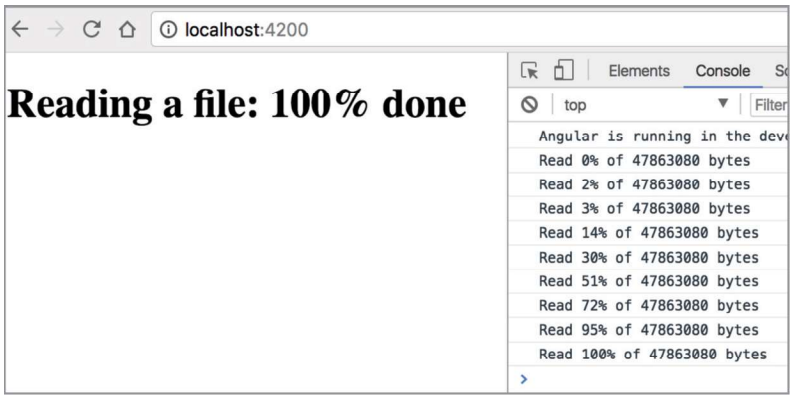


Figure 12.12 Reporting progress while reading a file

This app is located in the progressevents directory, and the content of app.component.ts is shown in the next listing.

Listing 12.29 app.component.ts

```
import {HttpClient, HttpEventType, HttpRequest} from '@angular/common/http';
import {Component} from "@angular/core";

@Component({
  selector: 'app-root',
  template: `<h1>Reading a file: {{percentDone}}% done</h1>
   `})
export class AppComponent{

  mydata: any;
  percentDone: number;

  constructor(private httpClient: HttpClient) {

    const req = new HttpRequest('GET',
                                './data/48MB_DATA.json',
                                {reportProgress: true});

    httpClient.request(req)
    .subscribe(data => {
      if (data.type === HttpEventType.DownloadProgress) {
        this.percentDone = Math.round(100 * data.loaded / data.total);))
        console.log(`Read ${this.percentDone}% of ${data.total} bytes`);
      } else {
```

Renders the current percentage

Declares an intention to make a GET request

Enables the progress event tracking

Specifies the file to read

Checks the type of the progress event

Calculates the current percentage

```
        this.mydata = data          <-- Emitted value is not a progress event
      }
    });
  }
}
```

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

```
ng serve --app progressevents -o
```

This app concludes our coverage of communicating with web servers using HTTP. In the next chapter, you'll see how an Angular client can communicate with web servers using WebSockets.

Summary

- Angular comes with the `HttpClient` service, which supports HTTP communications with web servers.
- Public methods of `HttpClient` return an `Observable` object, and only when the client subscribes to it is the request to the server made.
- An Angular client can communicate with web servers implemented in different technologies.
- You can intercept and replace HTTP requests and responses with modified ones to implement cross-cutting concerns.

13 Interacting with servers using the WebSocket protocol

This chapter covers

- Implementing a server data push to Angular clients
- Broadcasting data from the server to multiple clients
- Splitting ngAuction into two projects
- Implementing bidding in ngAuction

WebSocket is a low-overhead binary protocol supported by all modern web browsers (see https://en.wikipedia.org/wiki/WebSocket). It allows bidirectional message-oriented streaming of text and binary data between browsers and web servers. In contrast to HTTP, WebSocket is not a request/response-based protocol, and both server apps and client apps can initiate data push to the other party as soon as the data becomes available, in real time. This makes the WebSocket protocol a good fit for the following types of applications:

- Live trading/auctions/sports notifications
- Controlling medical equipment over the web
- Chat applications

- Multiplayer online games
- Real-time updates in social streams
- Live charts

All of these apps have one thing in common: there's a server (or a device) that may need to send an immediate notification to the user because some important event happened elsewhere. This is different from the use case when the user decides to send a request to the server for fresh data. For example, if a stock trade happens on the stock exchange, the notification has to be immediately sent to all users.

Another example is an online auction. If user Joe is considering bidding on a certain product, and user Mary (located 1,000 miles away) decides to increase the bid on the same product, you'd better push the notification to Joe right away as opposed to waiting until Joe refreshes the window.

We'll start this chapter with a brief comparison of HTTP and WebSocket protocols, and then we'll show you how a Node server can push data to a plain web page and to an Angular app.

In the hands-on section, you'll continue working on ngAuction. You'll start by splitting ngAuction into two projects: client and server. The server app will start two servers: the HTTP server will serve data, and the WebSocket server can receive user bids and push real-time bid notifications, emulating a scenario in which multiple users can bid on auctioned products. The Angular client interacts with both servers.

13.1 Comparing HTTP and WebSockets

With the request-based HTTP protocol, a client sends a request over a connection and waits for a response to come back, as shown in figure 13.1. Both the request and the response use the same browser-server connection. First, the request goes out, and then the response comes back via the same "wire." Think of a narrow bridge over a river where cars from both sides have to take turns crossing the bridge. In the web realm, this type of communications is called *half-duplex*.

The WebSocket protocol allows data to travel in both directions simultaneously (*full-duplex*) over the same connection, as shown in figure 13.2, and any party can initiate the data exchange. It's like a two-lane road. Another analogy is a phone conversation where two callers can speak and be heard at the same time. The WebSocket

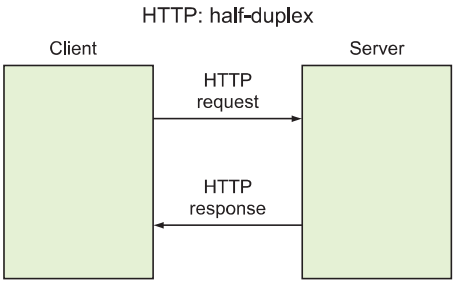


Figure 13.1 Half-duplex communication

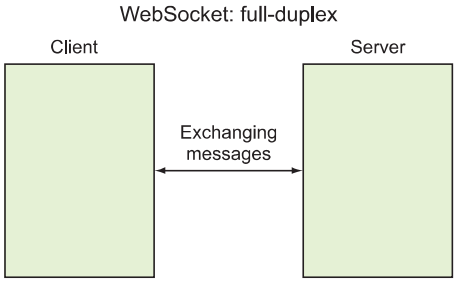


Figure 13.2 Full-duplex communication

connection is kept alive, which has an additional benefit: low latency in the interaction between the server and the client.

A typical HTTP request/response adds several hundred bytes (HTTP headers) to the application data. Say you want to write a web app that reports the latest stock prices every second. With HTTP, such an app would need to send an HTTP request (about 300 bytes) and receive a stock price that would arrive with an additional 300 bytes of an HTTP response object.

With WebSockets, the overhead is as low as a couple of bytes. Besides, there's no need to keep sending requests for the new price quote every second—this stock may not be traded for a while. Only when the stock price changes will the server push the new value to the client. Note the following observation (see goo.gl/zjj7Es):

> *Reducing kilobytes of data to 2 bytes is more than "a little more byte efficient," and reducing latency from 150 ms (TCP round trip to set up the connection plus a packet for the message) to 50 ms (just the packet for the message) is far more than marginal. In fact, these two factors alone are enough to make WebSocket seriously interesting to Google.*
>
> —Ian Hickson

NOTE Although most browsers support the binary protocol HTTP/2 (see https://http2.github.io)—which is more efficient than HTTP and also allows data push from the servers—it's not a replacement for the WebSocket protocol. The WebSocket protocol offers an API that allows pushing *data* to the client's app running in the browser, whereas HTTP/2 pushes *static resources* to the browser and is mainly for faster app delivery.

Every browser supports a `WebSocket` object for creating and managing a socket connection to the server (see http://mng.bz/1j4g). Initially, the browser establishes a regular HTTP connection with the server, but then your app requests a connection upgrade specifying the server's URL that supports the WebSocket connection. After that, the communication succeeds without the need of HTTP. The URLs of the WebSocket endpoints start with ws instead of http—for instance, ws://localhost:8085.

The WebSocket protocol is based on events and callbacks. For example, when your browser app establishes a connection with the server, it receives the `connection` event, and your app invokes a callback to handle this event. To handle the data that the server

may send over this connection, expect the `message` event providing the corresponding callback. If the connection is closed, the `close` event is dispatched so your app can react accordingly. In case of an error, the `WebSocket` object gets the `error` event.

On the server side, you'll have to process similar events. Their names may be different depending on the WebSocket software you use on the server. Let's write some code where a Node server will send data to the Angular app over WebSockets.

13.2 Pushing data from a Node server to a plain client

WebSockets are supported by most server-side platforms (Java, .NET, Python, and others). In chapter 12, you started working with Node servers, and you'll continue using Node for implementing your WebSocket server. In this section, you'll implement one particular use case: the server pushes data to a browser client as soon as the client connects to the socket. Since either party can start sending data over the WebSocket connection, you'll see that WebSockets aren't about request/response communication. Your simple client won't need to send a request for data—the server will initiate the communications.

Several Node packages implement the WebSocket protocol, and you'll use the npm package called `ws` (www.npmjs.com/package/ws). You can install this package and its type definitions by entering the following commands in your project directory:

```
npm install ws
npm install @types/ws --save-dev
```

The type definitions are needed so the TypeScript compiler won't complain when you use the API from the `ws` package. Besides, this file is handy for seeing the APIs and types available.

> **NOTE** The code that comes with this chapter has a directory called server, which contains the file package.json that lists both `ws` and `@types/ws` as dependencies. You just need to run the `npm install` command. Source code can be found at https://github.com/Farata/angulartypescript and www.manning.com/books/angular-development-with-typescript-second-edition.

Your first WebSocket server will be pretty simple: it'll push the text "This message was pushed by the WebSocket server" to an HTML/JavaScript client (no Angular) as soon as the socket connection is established. We purposely don't want the client to send any requests to the server so we can illustrate that a socket is a two-way street and that the server can push data without any request ceremony.

Your app creates two servers. The HTTP server runs on port 8000 and is responsible for sending an HTML page to the browser. When this page is loaded, it immediately connects to the WebSocket server that runs on port 8085. This server will push the message with the greeting as soon as the connection is established. The code of this app is located in the server/simple-websocket-server.ts file and is shown in the following listing.

Listing 13.1 simple-websocket-server.ts

```
import * as express from "express";
import * as path from "path";
import {Server} from "ws";

const app = express();

// HTTP Server
app.get('/', (req, res) => res.sendFile(path.join(__dirname,
                              '../simple-websocket-client.html')));

const httpServer = app.listen(8000, "localhost", () => {
     console.log(`HTTP server is listening on localhost:8000`);
});

// WebSocket Server
const wsServer = new Server({port: 8085});
 console.log('WebSocket server is listening on localhost:8085');

wsServer.on('connection',
           wsClient => {
              wsClient.send('This message was pushed by the WebSocket server');

              wsClient.onerror = (error) =>
                   console.log(`The server received: ${error['code']}`);
           }
);
```

You'll use Server from the ws module to instantiate a WebSocket server.

When the HTTP client connects with the root path, the HTTP server will send back this HTML file.

Starts the HTTP server on port 8000

Starts the WebSocket server on port 8085

Pushes the message to the newly connected client

Listens to connection event from clients

Handles connection errors

As soon as any client connects to your WebSocket server via port 8085, the connection event is dispatched on the server, and it'll also receive a reference to the object that represent this particular client's connection. Using the `send()` method, the server sends the greeting to this client. If another client connects to the same socket on port 8085, it'll also receive the same greeting.

> **NOTE** As soon as the new client connects to the server, the reference to this connection is added to the `wsServer.clients` array so you can broadcast messages to all connected clients if needed: `wsServer.clients.forEach (client => client.send('…'));`.

In your app, the HTTP and WebSocket servers run on different ports, but you could reuse the same port by providing the newly created `httpServer` instance to the constructor of the WebSocket server, as shown in the following listing.

Listing 13.2 Reusing the same port for both servers

```
const httpServer = app.listen(8000, "localhost", () => {...});

const wsServer = new Server({server: httpServer});
```

Creates an instance of the HTTP server on port 8000

Creates an instance of the WebSocket server based on the existing HTTP server

NOTE In the hands-on section, you'll reuse port 8000 for both HTTP and WebSocket communications (see the ng-auction/server/ws-auction.ts file).

The content of the server/simple-websocket-client.html file is shown in the next listing. This is a plain HTML/JavaScript client that doesn't use any frameworks.

Listing 13.3 simple-websocket-client.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
</head>
<body>
<span id="messageGoesHere"></span>

<script type="text/javascript">
    var ws = new WebSocket("ws://localhost:8085");

    ws.onmessage = function(event) {
         var mySpan = document.getElementById("messageGoesHere");
        mySpan.innerHTML = event.data;
    };

    ws.onerror = function(event) {
         console.log(`Error ${event}`);
    }
</script>
</body>
</html>
```

Establishes the socket connection

When the message arrives from the socket, displays its content in the element

In case of an error, the browser logs the error message on the console.

When the browser downloads this file, its script connects to your WebSocket server at ws://localhost:8085. At this point, the server upgrades the protocol from HTTP to WebSocket. Note that the protocol is `ws` and not `http`. For a secure socket connection, use the `wss` protocol.

To see this sample in action, run `npm install` in the server directory, compile the code by running the `tsc` command , and then start the server as follows:

```
node build/simple-websocket-server
```

You'll see the following messages on the console:

```
WebSocket server is listening on port 8085
HTTP server is listening on 8000
```

Open the Chrome browser and its Dev Tools at http://localhost:8000. You'll see the message, as shown on the left in figure 13.3. Under the Network tab on the right, you see two requests made to the server at localhost. The first one loads the simple-websocket-client.html file, and the second makes a request to the WebSocket that's open on port 8085 on your server.

In this example, the HTTP protocol is used only to initially load the HTML file. Then the client requests the protocol upgrade to WebSocket (status code 101), and

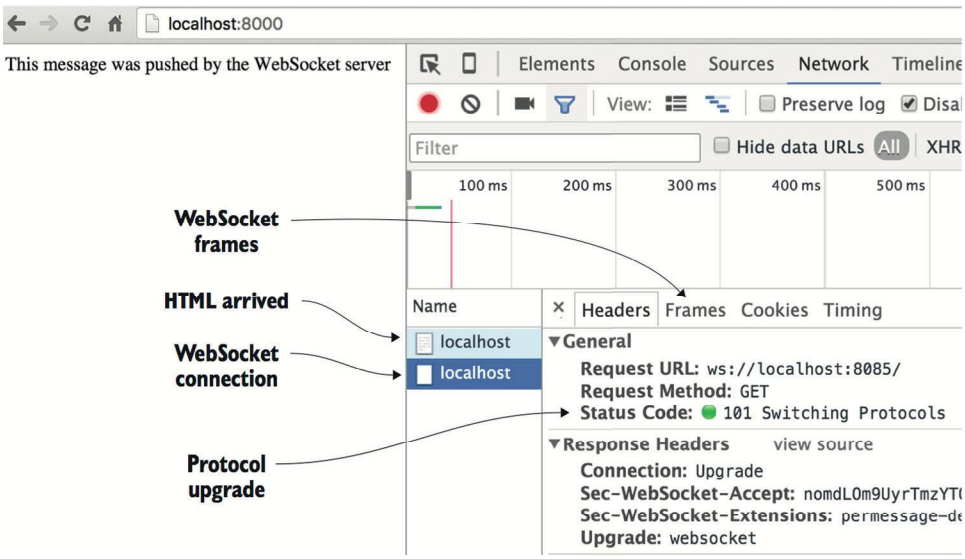


Figure 13.3 Getting the message from the socket

from then on this web page won't use HTTP. You can monitor data going over the socket using the Frames tab in Chrome Developer Tools. In this demo, you wrote a WebSocket client in JavaScript using the browser's native `WebSocket` object, but how can an Angular app consume or send messages to the server over WebSockets?

13.3 Using WebSockets in Angular clients

In Angular, you usually wrap all communications with servers into injectable services. In several apps in chapter 12, you did it with `HttpClient`, and you'll do it with the `WebSocket` object. But these two objects differ in that `HttpClient` is already an Angular injectable service that you'd *inject* into a service class of your app, whereas `WebSocket` is a native browser object, and you'll *create* it inside a service class.

There's another major difference between `HttpClient` and `WebSocket`. If making HTTP requests using `HttpClient` would return an observable with a single value, the `WebSocket` object offers an API that's easy to turn into an observable stream of multiple values like changing stock prices or bids on products.

Think of a WebSocket as a data producer that can emit values, and an `Observable` object can relay these values to subscribers (for example, Angular components). In Angular, you can either manually create a service producing an observable stream from a WebSocket connection or use the `WebSocketSubject` offered by RxJS. In this chapter, you'll see both ways of handling WebSocket messages in Angular clients.

But first, let's see how to wrap any `Observable` emitting values into an Angular service.

13.3.1 Wrapping an observable stream into a service

In this section, you'll create an observable service that emits hardcoded values without connecting to any server. In section D.5 in appendix D, we explain how to use the `Observable.create()` method, providing an observer as an argument. If you haven't read appendix D yet, do it now.

The following listing creates a service with a method that takes an observer as an argument and emits the current time every second.

Listing 13.4 observable.service.ts

```
import {Observable} from 'rxjs';

export class ObservableService {

  createObservableService(): Observable<Date> {

    return new Observable(
          observer => {
              setInterval(() =>
                 observer.next(new Date())
              , 1000);
          }
      );
  }
}
```

Returns an observable stream of dates

Creates an observable

Provides an observer

Emits the new date every second

In this service, you create an instance of the `Observable` object, assuming that the subscriber will provide an `Observer` that knows what to do with the emitted data. Whenever the observable invokes the `next(new Date())` method on the observer, the subscriber will receive the current date and time. Your data stream never throws an error and never completes.

You'll inject the `ObservableService` into the `AppComponent`, which invokes the `createObservableService()` method and subscribes to its stream of values, creating an observer that knows what to do with data. The observer just assigns the received time to the `currentTime` variable that renders the time on the UI, as shown in the following listing.

Listing 13.5 observableservice/app.component.ts

```
import {Component} from "@angular/core";
import {ObservableService} from "./observable.service";

@Component({
  selector: 'app-root',
  providers: [ObservableService],
  template: `<h1>Custom observable service</h1>
       Current time: {{currentTime | date: 'mediumTime'}}
   `})
export class AppComponent {

  currentTime: Date;
```

Displays the time using the date pipe

```
constructor(private observableService: ObservableService) {
    this.observableService.createObservableService()
       .subscribe(data => this.currentTime = data);
  }
}
```

Injects the service that wraps the observable

Creates the observable and starts emitting dates

Subscribes to the stream of dates

This app doesn't use any servers, and you can see it in action here. Run it by entering the following command in the client directory (after `npm install`):

```
ng serve --app observableservice -o
```

In the browser window, the current time will be updated every second. You use the `DatePipe` here with the format `'mediumTime'`, which displays only hours, minutes, and seconds (all date formats are described in the Angular `DatePipe` documentation at http://mng.bz/78lD).

This simple example demonstrates a basic technique for creating an injectable service that wraps an observable stream so components or user services can subscribe to it. In this case, you use `setInterval()`, but you could replace it with any application-specific code that generates one or more values and emits them as a stream.

Don't forget about error handling and completing the stream if need be. The following listing shows an observable that sends one element to the observer, may throw an error, and notifies the observer that streaming is complete.

Listing 13.6 Sending errors and completion events

```
return new Observable(
    observer => {
      try {
        observer.next('Hello from observable');

        // throw("Got an error");
         // some other code can be here
      } catch(err) {
         observer.error(err);
       } finally {
         observer.complete();
       }
    }
);
```

Sends the text value to the observer

Emulates an error situation

Sends the error to the observer

Always let the observer know that the data streaming is over.

If you uncomment the line with `throw`, the preceding program will jump over "some other code" and continue in the `catch` section, where you invoke `observer.error()`. This will result in the invocation of the error handler on the subscriber, if there is one.

The data producer for your observable stream was the time generator, but it could be a WebSocket server generating some useful values. Let's create an Angular service that communicates with a WebSocket server.

13.3.2 Angular talking to a WebSocket server

In the hands-on section, you'll implement a real-world use case of an Angular client communicating with a server over WebSockets. This is how users of ngAuction will place bids and receive notifications of bids made by other users.

In this section, we'll show you a very basic way to wrap a `WebSocket` into an Angular client. This is going to be a rather simple wrapper for the `WebSocket` object, but in the hands-on section, you'll use a more robust `WebSocketSubject` that comes with RxJS.

Your next Angular app will include a service that interacts with the Node WebSocket server. The server-side tier can be implemented with any technology that supports WebSockets. Figure 13.4 illustrates the architecture of such an application (think of bidding messages going between the client and server over the socket connection).

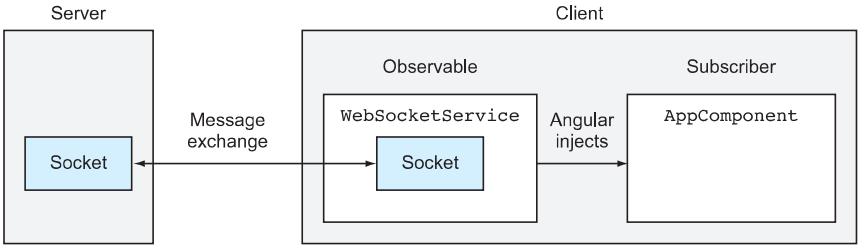


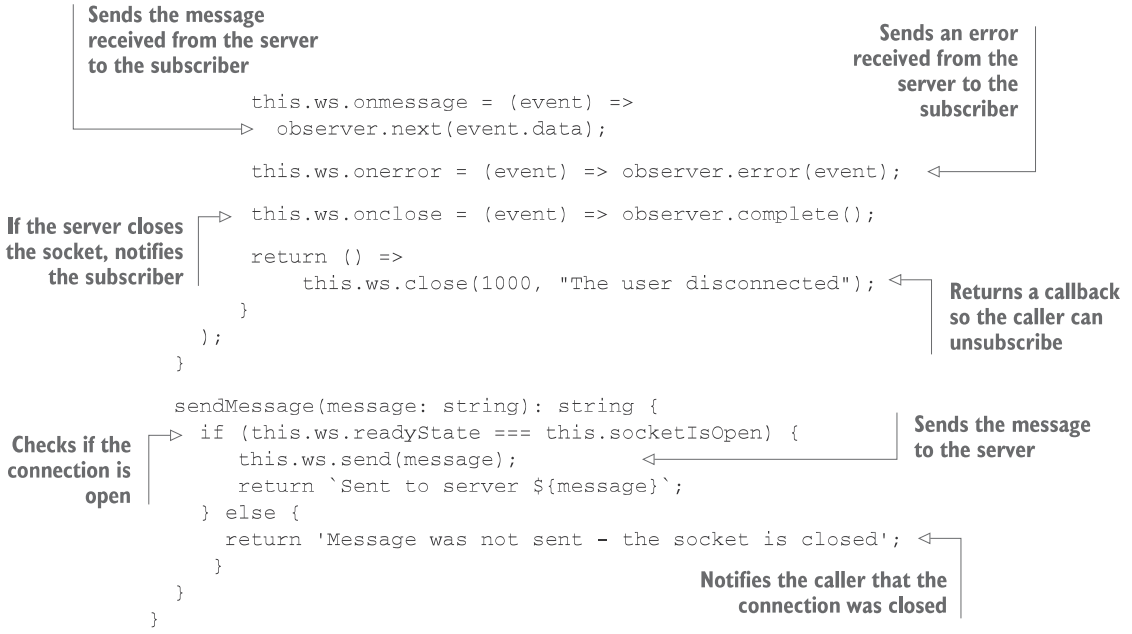
Figure 13.4 Angular interacting with a server via a socket

The code in listing 13.7 wraps the browser's `WebSocket` object into an observable stream. This service creates an instance of the `WebSocket` object that's connected to the WebSocket server based on the provided URL, and the client instance handles messages received from the server.

Your `WebSocketService` also has a `sendMessage()` method so the client can send messages to the server as well. Prior to sending the message, the service checks whether the connection is still open (the `WebSocket.readyState === 1`), as shown in the following listing.

Listing 13.7 wsservice/websocket.service.ts

```
import {Observable } from 'rxjs';

export class WebSocketService {

  ws: WebSocket;
  socketIsOpen = 1;      <-- The WebSocket is open.

  createObservableSocket(url: string): Observable<any> {   <-- This method emits messages received from the specified URL.
     this.ws = new WebSocket(url);                          <-- Connects to the WebSocket server

    return new Observable(                                  <-- Creates an Observable object
       observer => {
```



Note that your observer returns a callback, so if the caller invokes the `unsubscribe()` method, this callback will be invoked. It'll close the connection, sending a 1000 status code and the message explaining the reason for closing. You can see all allowed codes for closing the connection at http://mng.bz/5V07.

Now let's write the `AppComponent` that subscribes to the `WebSocketService`, which is injected into the `AppComponent` shown in figure 13.4. This component, shown in the following listing, can also send messages to the server when the user clicks the Send Message to Server button.

Listing 13.8 wsservice/app.component.ts

```
import {Component, OnDestroy} from "@angular/core";
import {WebSocketService} from "./websocket.service";
import {Subscription} from "rxjs";

@Component({
  selector: 'app-root',
  providers: [ WebSocketService ],
  template: `<h1>Angular client for a WebSocket server</h1>
  {{messageFromServer}}<br>
  <button (click)="sendMessageToServer()">Send Message to Server</button>
  <button (click)="closeSocket()">Disconnect</button>
  <div>{{status}}</div>
  `})
export class AppComponent implements OnDestroy {

  messageFromServer: string;
  wsSubscription: Subscription;
   status;
```

This property will hold the reference to the subscription.

```
  constructor(private wsService: WebSocketService) {   <-- Injects the service
    this.wsSubscription =
      this.wsService.createObservableSocket("ws://localhost:8085")
       .subscribe(
        data => this.messageFromServer = data,    <-- Handles the data received from the server
         err => console.log( 'err'),
         () =>  console.log( 'The observable stream is complete')
       );
  }

  sendMessageToServer(){                                   Sends the message to the server
    this.status = this.wsService.sendMessage("Hello from client");   <--
   }

  closeSocket(){
    this.wsSubscription.unsubscribe();      <-- Closes the WebSocket connection
     this.status = 'The socket is closed';
  }

  ngOnDestroy() {
    this.closeSocket();
  }
}
```

Connects to the server

Note that you store the reference to the subscription in the `wsSubscription` property, and when the user clicks the Disconnect button, this component unsubscribes from the observable. That invokes the callback defined in the observer, closing the connection to the WebSocket.

The client is ready. Now we'll show you the code for the server that will communicate with this client. The callback function that's invoked on the connection event sends the greeting to the client and adds two more event handler functions to the object that represents this particular client.

One function handles messages received from the client, and another handles errors (you'll log the error code). This server is implemented in the two-way-websocket-server.ts file.

Listing 13.9 server/two-way-websocket-server.ts

```
import {Server} from "ws";
let wsServer = new Server({port:8085});   <-- Starts the WebSocket serve

console.log('WebSocket server is listening on port 8085');

wsServer.on('connection',    <-- A new client connected
    websocket => {

    websocket.send('Hello from the two-way WebSocket server');

    websocket.onmessage = (message) =>
             console.log(`The server received: ${message['data']}`);
});
```

Greets the newly connected client

Listens to the message from this client

```
    websocket.onerror = (error) =>
        console.log(`The server received: ${error['code']}`);

    websocket.onclose = (why) =>
        console.log(`The server received: ${why.code} ${why.reason}`);
});
```

Logs the error from this connection, if any

The client disconnected, so you log the reason.

To see this app in action, start the server by running the following command from the server directory:

```
node build/two-way-websocket-server
```

Then build and start the Angular app from the client directory as follows:

```
ng serve --app wsservice
```

To emulate a scenario where more than one client is connected to the same WebSocket server, open two browsers at http://localhost:4200. Each of the apps will receive a greeting from the server, and you'll be able to send messages to the server by clicking the Send Message to Server button.

We took the screenshot in figure 13.5 after the button is clicked once (Chrome Developer Tools has the WS and Frames tabs opened under Network). On the right, you see the greeting message that arrived from the server and the message that the client sent to the server.

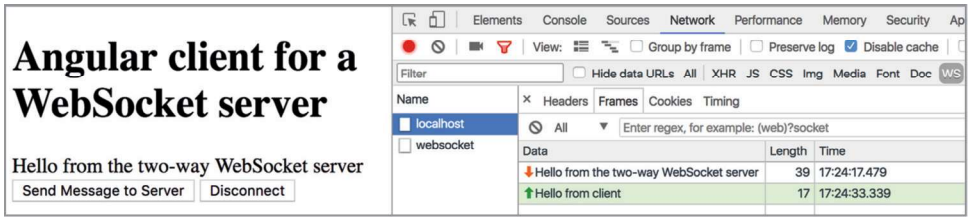


Figure 13.5 Getting the message in Angular from Node

Figure 13.6 shows the screenshot taken after the client clicks the Send Message to Server button, then Disconnect, and then Send Message to Server again.

> **NOTE** Browsers don't enforce the same-origin policy on WebSocket connections. That's why you're able to exchange data between the client originating from port 4200 and the server running on port 8085. Refer to the documentation of whatever server-side technology you use to see what protection is available for WebSockets.

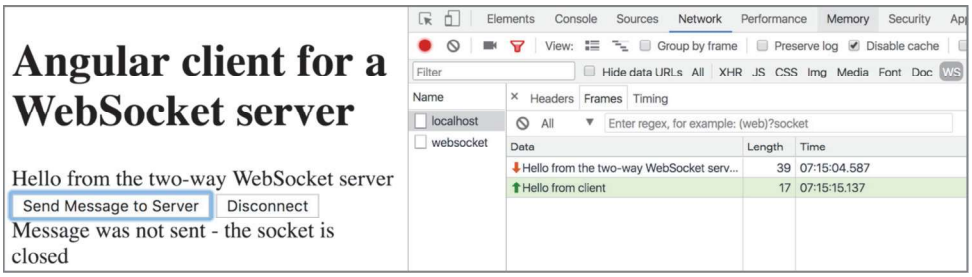


Figure 13.6 Send, disconnect, and send again

Integrating WebSockets with server-side messaging systems

Imagine your server uses a messaging system; let's use ActiveMQ as an example. Say you'd like to enable your JavaScript client to exchange data with ActiveMQ over WebSockets. If you decide to program such data exchange from scratch, you need to come up with a way to notify the server's endpoint that the data sent by the client should be redirected into an ActiveMQ queue with a specific name. Then the server-side code needs to format the client's message to be accepted by ActiveMQ according to its internal protocol. Also, the server-side app needs to keep track of all connected clients, and possibly implement some heartbeats to monitor the health of the socket connection. That's a lot of coding.

The good news is that WebSockets can use subprotocols to integrate with server-side messaging systems. For example, server-side code can map the WebSocket endpoint to an existing queue in ActiveMQ. This way, when a server's software places a message into a queue, it's automatically pushed to the client. Similarly, when a client sends a message to a WebSocket endpoint, it's placed in the queue on the server. Implementing heartbeats comes down to providing a configuration option.

STOMP is one of the popular subprotocols used for sending text messages over WebSockets (see http://mng.bz/PPsy). It describes a client-side message broker that communicates with its server-side peer. For client STOMP support, we use ng2-stompjs, available at http://mng.bz/KdIM.

The server-side admin should install a STOMP connector for their messaging server (ActiveMQ has native STOMP support). In such a setup, client-server communication is more robust and requires less coding on the application level.

In chapter 12, you learned how to communicate with a web server via HTTP. In this chapter, we introduced the WebSocket protocol. The next version of ngAuction will use both communication protocols, but first let's see how the materials covered in this chapter apply to the new functionality of ngAuction that you're about to implement.

The WebSocket protocol isn't based on the request/response model, and the WebSocket server can initiate the communication with the client without any additional ceremony. This is a valuable feature for ngAuction, because the server knows first when any users place a bid on each auctioned product in this multi-user app. Because the server doesn't need to wait for the client's requests for data, it can push the newly placed bids to all users that are connected to this WebSocket server. That means the server can push the latest bids to all users in real time.

13.4 Hands-on: Node server with WebSockets support

In this section, we'll review the refactored version of ngAuction that comes with this chapter. In real auctions, multiple users can bid on products. When the server receives a bid from a user, the bid server should broadcast the latest bid to all users who are watching selected products. This version of ngAuction accomplishes the following main tasks:

- Split ngAuction into two separate projects, client and server, and store the product data and images on the server.
- Modify the client so it'll use the `HttpClient` service to make requests to the server to get products data.
- On the server side, implement HTTP and WebSocket servers. The HTTP server will serve product data.
- The WebSocket server will accept user bids on selected products, and all other users can see the latest bids pushed by the server.

Figure 13.7 shows the rendered `ProductDetailComponent` with the button that will allow a user to place bids in $5 increments. If a user clicks this button once, the price will change to $75 on their UI, as well as for all other users having the product-detail view open for the same product. The server will broadcast (via a WebSocket connection) the latest bid amounts to all users who are looking at this product.

To implement this functionality, you'll add WebSocket support to the server and create a new `BidService` on the client. Figure 13.8 shows the main players involved in client-server communications in this version of ngAuction.

The *DI* in figure 13.8 stands for *dependency injection*. Angular injects the `HttpClient` service into `ProductService`, which in turn is injected into three components: `CategoriesComponent`, `SearchComponent`, and `ProductComponent`. `ProductService` is responsible for all HTTP-based communications with the server.

The `BidService` wraps all WebSocket-based communications with the server. It's injected into the `ProductDetailComponent`. When a user opens the product-detail view, the new bids (if any) will be displayed. When a user places a new bid, the `BidService` will push the message to the server. When the WebSocket server pushes a new bid, the `BidService` receives the bid, and the `ProductDetailComponent` renders it.

Figure 13.8 shows two more injectable values: `API_BASE_URL` and `WS_URL`. The former will contain the URL of the HTTP server, and the latter, the URL of the

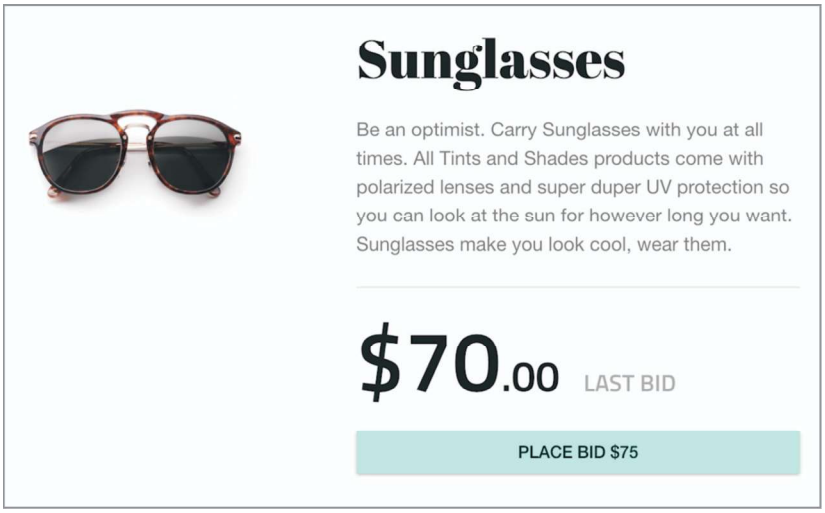


Figure 13.7 The `ProductDetailComponent` with a bid button

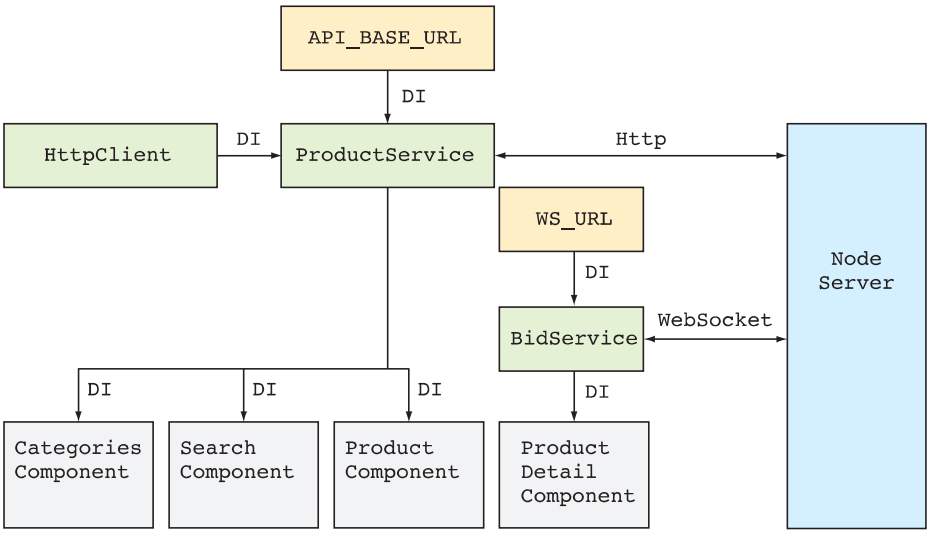


Figure 13.8 Client-server communications in ngAuction

WebSocket server. To inject these values, you'll use `InjectionToken`. Both URLs are configurable, and their values are stored in the Angular project in the environments/environment.ts and environments/environment.prod.ts files.

The environment.ts file is used in dev mode and is shown in the following listing.

Listing 13.10 environment.ts

```
export const environment = {
  production: false,
  apiBaseUrl: 'http://localhost:9090/api',
  wsUrl: 'ws://localhost:9090'
};
```

The environment.prod.ts file is used in production mode, and because the Angular app is expected to be deployed on the same server that serves data, there's no need to specify the full URL for HTTP communications, as shown in the following listing.

Listing 13.11 environment.prod.ts

```
export const environment = {
  production: true,
  apiBaseUrl: '/api',
  wsUrl: 'ws://localhost:9090'
};
```

13.4.1 Running ngAuction in dev mode

ngAuction consists of two projects now, so you need to run `npm install` in each project, and then start the server and the client separately. To start the server, change to the server directory, compile all the TypeScript code into JavaScript by running `tsc`, and start the server as follows:

```
node build/main
```

To start the Angular app, go to the client directory and run the following command:

```
ng serve
```

You'll see the same UI of ngAuction that you created in chapter 11, but now product data and images come from the server via the HTTP connection. Open your Chrome browser at http://localhost:4200, select a product, and click the bid button. You'll see how the price increases by $5. Now open another browser (such as Firefox) at http://localhost:4200, select the same product, and you'll see the latest price. Place a new bid in that second browser, and the new bid is shown in both browsers. The server pushes the new bid to all connected clients.

After reading in chapter 12 about the same-origin restriction and proxying client requests, you may be wondering how the app loaded from port 4200 can access data on the HTTP server running on port 9090 without configuring the proxy on the client. It's because this time, you used a special CORS package on the Node server for unrestricted access from any client. You'll see how to do this in the next section.

13.4.2 Reviewing the ngAuction server code

By now, you know how to create and start HTTP and WebSocket servers using Node.js, Express, and the `ws` package, and you won't repeat that part. In this section, we'll review the code fragments of the server that are relevant to the new functionality of ngAuction. You'll split the server code into four TypeScript files. Figure 13.9 shows the structure of the ngAuction server directory that comes with this chapter.

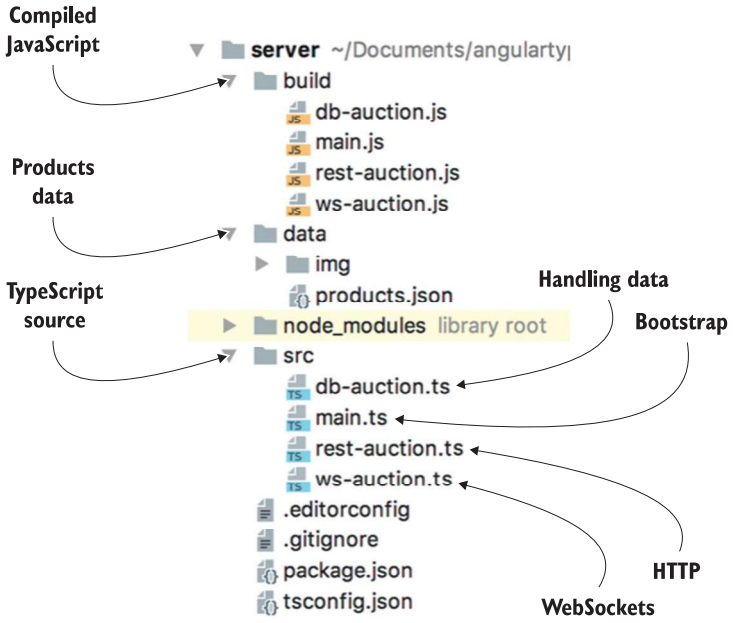


Figure 13.9 ngAuction server structure

In ngAuction from chapter 11, the data folder was located in the Angular project; now, you move the data to the server. In chapter 11, the code that read products.json and the functions to get all the products or products by ID was located in the product .service.ts file, and now is located in the db-auction.ts file. The main.ts file contains the code to launch both HTTP and WebSocket servers. The ws-auction.ts file has the code supporting WebSocket communication with ngAuction clients.

Launching the HTTP and WebSocket servers

Let's start the code review from the main.ts file that's used to launch the servers. The code in this file is similar to simple-websocket-server.ts from section 13.2, but this time you don't start HTTP and WebSocket servers on different ports—they both use port 9090. Another difference is that you create an instance of the HTTP server using the `createServer()` function, one of Node's interfaces (see https://nodejs.org/api/http.html), as shown in the following listing.

Listing 13.12 main.ts

```
import * as express from 'express';
import {createServer} from 'http';
import {createBidServer} from './ws-auction';
import {router} from './rest-auction';

const app = express();
app.use('/api', router);

const server = createServer(app);
 createBidServer(server);

server.listen(9090, "localhost", () => {
   const {address, port} = server.address();
  console.log(`Listening on  ${address} ${port}`);
});
```

Forwards requests containing /api to the Express router

Creates an instance of the http.Server object

Creates an instance of BidServer using its HTTP peer

Launches both servers

This code creates an instance of the HTTP server with Node's `createServer()`, and you pass Express as a callback function to handle all HTTP requests. To start your WebSocket server, you invoke the `createBidServer()` function from ws-auction.ts. But first, let's review your RESTful HTTP server.

THE HTTP SERVER

In chapter 12, section 12.2.2, you created a simple Node/Express server that handles requests for products. In this section, you'll see a more advanced version of such a server. Here, you'll use the Express `Router` to route HTTP requests. You'll also use the CORS module to allow requests from all browsers to ignore the same-origin restriction. That's why you can start the client using `ng serve` without the need to configure a proxy.

Finally, the product data won't be hardcoded—you moved the data-handling part into the db-auction.ts script. Your HTTP REST server is implemented in the rest-auction.ts file, shown in the following listing.

Listing 13.13 rest-auction.ts

```
import * as cors from 'cors';
 import * as express from 'express';
import {
  getAllCategories,
  getProducts,
  getProductById,
  getProductsByCategory
} from './db-auction';

export const router = express.Router();

router.use(cors());

router.get('/products',
  async (req: express.Request, res: express.Response) => {
   res.json(await getProducts(req.query));
 });
```

Imports the CORS module

Imports the data-handling functions

Creates and exports the Express Router instance

Uses CORS to allow requests from all clients

Uses the async keyword to mark the function as asynchronous

Uses the await keyword to avoid nesting code in then() callbacks

```
router.get('/products/:productId', async (req: express.Request,
➥res: express.Response) => {
  const productId = parseInt(req.params.productId, 10) || -1;
  res.json(await getProductById(productId));
});

router.get('/categories', async (_, res: express.Response) => {
  res.json(await getAllCategories());
});

router.get('/categories/:category', async (req: express.Request,
➥res: express.Response) => {
  res.json(await getProductsByCategory(req.params.category));
});
```

In section 13.4.1, we stated that the dev server on the client will run on port 4200, and the REST server will run on port 9090. To overcome the same-origin restriction, you use the Express package CORS to enable access from all origins (see http://mng.bz/aNxM). If you open package.json in the server directory, you'll find the dependency `"cors": "^2.8.4"` there.

In this server, you create an instance of the Express `Router` object and use it to route HTTP GET requests based on the provided path.

Note the use of the `async` and `await` keywords. You didn't use them for product retrieval in chapter 12, section 12.2.2, because product data was stored in an array, and functions like `getProducts()` were synchronous there. Now you use the data-handling functions from db-auction.ts, and they read data from a file, which is an asynchronous operation.

Using the `async` and `await` keywords makes the async code look as if it's synchronous (see section A.12.4 in appendix A for more details).

THE DATA-HANDLING SCRIPT

Your HTTP server uses the db-auction.ts script for all data-handling operations. This script has methods to read products from the products.json file as well as search products based on provided search criteria. We won't be reviewing the entire code of the db-auction.ts script, but we will discuss the code changes compared to product.service.ts from the version of ngAuction included with chapter 11, as shown in the following listing.

Listing 13.14 db-auction.ts (partial listing)

```
import * as fs from 'fs';
import * as util from 'util';

type DB = Product[];

const readFile = util.promisify(fs.readFile);
 const db$: Promise<DB> =
   readFile('./data/products.json', 'utf8')
   .then(JSON.parse, console.error);
```

Defines a new type to store an array of products

Makes fs.readFile to return a promise

Declares a promise for reading products.json

Reads products.json

```
export async function getAllCategories(): Promise<string[]> {
  const allCategories = (await db$)
    .map(p => p.categories)
    .reduce((all, current) => all.concat(current), []);

  return [...new Set(allCategories)];
}
...

export async function updateProductBidAmount(productId: number,
                                              price: number): Promise<any> {
  const products = await db$;
  const product = products.find(p => p.id === productId);
  if (product) {
    product.price = price;
  }
}
```

This function gets category names for each product.

Gets rid of duplicate categories

This function updates the product price based on the latest bid.

In chapter 11, the products.json file was located on the client side, and `ProductService` read this file using the `HttpClient` service, as follows:

```
http.get<Product[]>('/data/products.json');
```

Now this file is located on the server, and you read it using Node's `fs` module, which includes functions for working with the filesystem (see https://nodejs.org/api/fs.html). You also use another Node module, `util`, that includes a number of useful utilities, and you use `util.promisify()` to read the file returning the data as a promise (see http://mng.bz/Z009) instead of providing a callback to `fs.readFile`.

In several places in db-auction.ts, you see `await db$`, which means "execute the `db$` promise and wait until it resolves or rejects." The `db$` promise knows how to read the products.json file.

Now that we've discussed how your RESTful server works, let's get familiar with the code of the WebSocket server.

THE WEBSOCKET SERVER

The ws-auction.ts script implements your WebSocket server that can receive bids from users and notify users about new bids. A bid is represented by a `BidMessage` type containing the product ID and the bid amount (price), as shown in the following listing.

Listing 13.15 `BidMessage` from ws-auction.ts

```
interface BidMessage {
  productId: number;
  price: number;
}
```

The `createBidServer()` function creates an instance of the class `BidServer`, using the provided instance of `http.Server`, as shown in the next listing.

Listing 13.16 `createBidServer()` from ws-auction.ts

```
export function createBidServer(httpServer: http.Server): BidServer {
  return new BidServer(httpServer);
}
```

The `BidServer` class contains the standard WebSocket callbacks `onConnection()`, `onMessage()`, `onClose()`, and `onError()`. The constructor of this class creates an instance of `ws.Server` (you use the `ws` package there) and defines the `onConnection()` callback method to the WebSocket `connection` event. The outline of the `BidServer` class is shown in the following listing.

Listing 13.17 The structure of `BidServer`

```
export class BidServer {
  private readonly wsServer: ws.Server;

  constructor(server: http.Server) {}

  private onConnection(ws: ws): void {...}

  private onMessage(message: string): void {...}

  private onClose(): void {...}

  private onError(error: Error): void {...}
 }
```

Handler for the WebSocket connection event

Handler for the message event

Handler for the close event

Handler for the error event

Now let's review the implementation of each method, starting with `constructor()`

Listing 13.18 The constructor

```
constructor(server: http.Server) {
  this.wsServer = new ws.Server({ server });
   this.wsServer.on('connection',
      (userSocket: ws) => this.onConnection(userSocket));
}
```

Instantiates the WebSocket server using the HTTP server instance

Defines the handler of the connection event

When your ngAuction client connects to `BidServer`, the `onConnection()` callback is invoked. The argument of this callback is the `WebSocket` object representing a single client's connection. When the client makes the initial request to switch the protocol from HTTP to `WebSocket`, it'll invoke the `onConnection()` callback, shown in the following listing.

Listing 13.19 Handling the connection event

```
private onConnection(ws: ws): void {
  ws.on('message', (message: string) => this.onMessage(message));
   ws.on('error', (error: Error) => this.onError(error));
```

Listens to message events

Listens to error events

```
    ws.on('close', () => this.onClose());  <-- Listens to close events
  console.log(`Connections count: ${this.wsServer.clients.size}`);
}
```

Reports the number of connected clients

The `onConnection()` method assigns the callback methods for the WebSocket events `message`, `close`, and `error`. When the `ws` module creates an instance of the WebSocket server, it stores the references to connected clients in the `wsServer.clients` property. On every connection, you print on the console the number of connected clients. The next listing reviews the callback methods one by one, starting from `onMessage()`.

Listing 13.20 Handling the client messages

```
import { updateProductBidAmount} from './db-auction';
...
  private onMessage(message: string): void {
    const bid: BidMessage = JSON.parse(message);
     updateProductBidAmount(bid.productId, bid.price);

    // Broadcast the new bid
    this.wsServer.clients.forEach(ws => ws.send(JSON.stringify(bid)));

    console.log(`Bid ${bid.price} is placed on product ${bid.productId}`);
  }
```

Parses the client's BidMessage

Updates the bid amount in your in-memory database

Sends new product bid information to all subscribers

The `onMessage()` callback gets the user's bid on the product and updates the amount in your simple in-memory database implemented in the db-auction.ts script. If a user opens the product-detail view, they become a subscriber to notifications about all users bids, so you push the new bid over the socket to each subscriber.

Next, we'll review the callback for the `close` event. When a user closes the product-detail view, the WebSocket connection is closed as well. In this case, the closed connection is removed from `wsServer.clients`, so no bid notifications will be sent to a nonexistent connection, as shown in the following listing.

Listing 13.21 Handling closed connections

```
private onClose(): void {
  console.log(`Connections count: ${this.wsServer.clients.size}`);
}
```

In the `onError()` callback, you extract the error message from the provided `Error` object and log the error on the console.

Listing 13.22 Handling WebSocket errors

```
private onError(error: Error): void {
    console.error(`WebSocket error: "${error.message}"`);
  }
}
```

Figure 13.10 shows the same product-detail view open in Chrome and Firefox browsers. The latest bid is synchronized in both views as soon as the user clicks the bid button in any of the browsers.

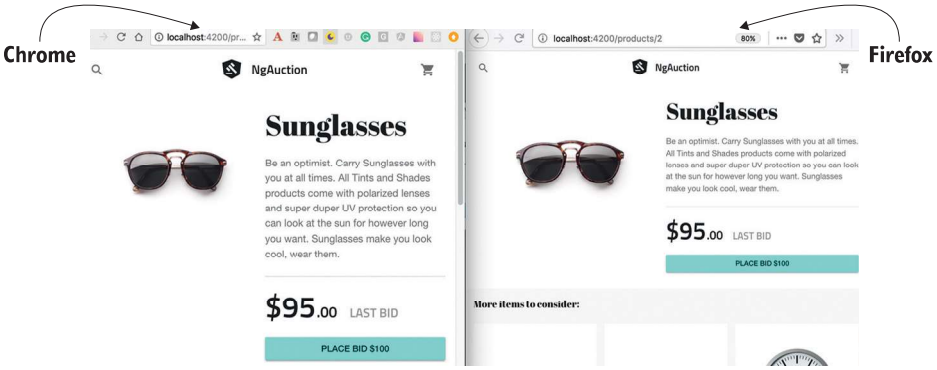


Figure 13.10 Synchronized bids in two browsers

Homework

Your ngAuction isn't a production-grade auction, and you may find some edge cases that aren't properly handled. We'll describe one of them in case you want to improve this app.

Imagine that the current bid of the product is $70, and Joe clicks the bid button to make a $75 bid. At the same time, Mary, who was also shown $70 as the latest product bid, also clicks the bid button. There could be a situation when Joe's request will change the bid to $75 on the server, and some milliseconds later, Mary's $75 bid arrives to the server. Currently, the `BidServer` will just replace Joe's $75 with Mary's $75, and each of them will assume that he or she placed a $75 bid.

To prevent this from happening, modify the code in the `BidServer` to reject the bid unless the bid amount is greater than the existing one. In such scenarios, send the losing user a message containing the new minimum bid amount.

We've covered the server-side code of ngAuction; let's see what changed on the client side compared to the version in chapter 11.

13.4.3 What changed in the ngAuction client code

As stated in the previous section, the main change in the Angular code of ngAuction is that you moved to the server the file with product data and images, and the code that reads these files. Accordingly, you added the code to `ProductService` to interact with the HTTP server using the `HttpClient` service covered in chapter 12.

Remember that the `ProductComponent` is responsible for rendering the product view that includes `ProductDetailComponent` and `ProductSuggestionComponent` with the grid of suggested products, and you modified the code of both components.

Also, you added the bid.service.ts file to communicate with the WebSocket service and modified the code in product-detail.component.ts so the user can place bids on a product and see other user bids. Let's review the changes related to the product view.

TWO OBSERVABLES IN THE PRODUCTCOMPONENT

First, the product view would change if the user looks at the details of one product and then selects another one from the suggested products grid. The route doesn't change (the user is still looking at the product view), and `ProductComponent` subscribes to an observable from `ActivatedRoute` that emits the newly selected `productId` and retrieves the corresponding product details, as shown in the following listing.

Listing 13.23 Handling changed parameters from `ActivatedRoute`

```
this.product$ = this.route.paramMap
  .pipe(
    map(params => parseInt(params.get('productId') || '', 10)),
    filter(productId => Boolean(productId)),
    switchMap(productId => this.productService.getById(productId))
  );
```

Gets the new productId

Handles the possibly invalid productId.

Retrieves the selected product details

In listing 13.23, you retrieve `productId` from `ActivatedRoute` and pass it over to the `ProductService`, using the `switchMap` operator. The `filter` operator is just a precaution to weed out the falsy product IDs. For example, a user can manually enter an erroneous URL like http://localhost:4200/products/A23, and you don't want to request the details for a nonexistent product.

The template of the product component includes the `<nga-product-detail>` component, which gets a selected product via its input property `product`, as shown in the following listing.

Listing 13.24 Passing a selected product to `ProductDetailComponent`

```
<nga-product-detail
    fxFlex="auto"
    fxFlex.>-md="65%"
    *ngIf="product$ | async as product"
     [product]="product">
 </nga-product-detail>
```

Extracts the Product object with async pipe

Passes the selected Product to the ProductDetailComponent

You place the code that unwraps product data inside the `*ngIf` directive because the product data is retrieved asynchronously, and you want to make sure that the `product$` observable emitted the data that you bind to the input property of the `ProductDetailComponent`. Let's see how the `ProductDetailComponent` handles the received product.

PLACING AND MONITORING BIDS IN PRODUCTDETAILCOMPONENT

The UI of the `ProductDetailComponent` is shown in figure 13.10. This component gets the product to display via its input property `product`. If a user clicks the bid button, the new bid ($5 more than the current one) is sent over the WebSocket connection using `BidService`, which implements all communications with the `BidServer`. If another user connected to the `BidServer` bids on the same product, the bid amount on the product-detail view will be immediately updated.

The `ProductDetailComponent` class has the private RxJS subject `productChange$` and the observable `latestBids$`, which merges the data of two observables:

- `productChange$` handles the case when a user opens the product-detail view and then selects another product from the list "More items to consider." When the binding to the input parameter `product` changes, the lifecycle hook `ngOnChanges()` intercepts the change, and `productChange$` emits the product data.
- `latestBids$` emits the new value when either `productChange$` or `BidService` pushes the new bid received from the server.

You have two data sources that can emit values, and on any emission you need to update the view. That's why you combine two observables with the RxJS operator `combineLatest`. The code of the product-detail.component.ts file is shown in the following listing.

Listing 13.25 product-detail.component.ts

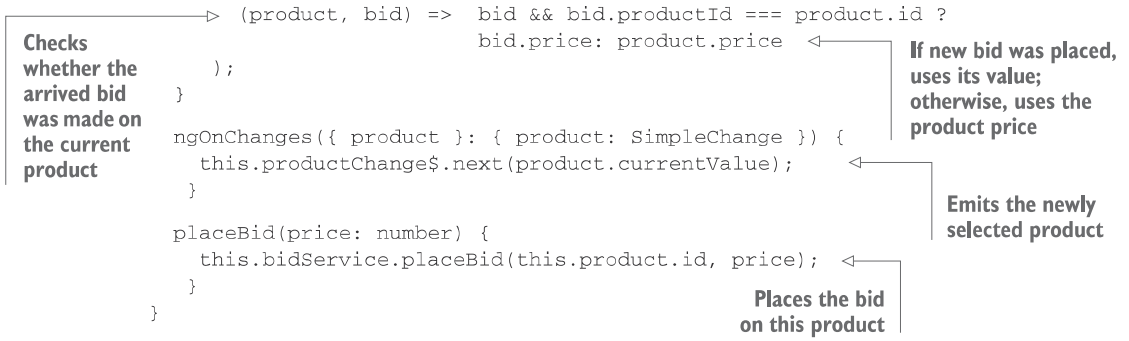
```
// imports are omitted for brevity
@Component({
  selector: 'nga-product-detail',
  styleUrls: [ './product-detail.component.scss' ],
  templateUrl: './product-detail.component.html',
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class ProductDetailComponent implements OnInit, OnChanges {
  private readonly productChange$ = new Subject<Product>();
  latestBids$: Observable<number>;
  @Input() product: Product;

  constructor(private bidService: BidService) {}

  ngOnInit() {
    this.latestBids$ = combineLatest(
      this.productChange$.pipe(startWith(this.product)),
      this.bidService.priceUpdates$.pipe(startWith<BidMessage | null>(null)),
```

Injects the BidService

Combines the values of two observables

The second observable emits bids.

The first observable starts emission with the currently displayed product.



The RxJS operator `combineLatest` (see http://mng.bz/Y28Y) subscribes to the values emitted by two observables and invokes the merge function when either of the observables emits the value. In this case, it's either the value emitted by `productChange$` or by `bidService.priceUpdates$` (the `BidService` code is included in the next section). Here's your merge function:

```
(product, bid) =>  bid && bid.productId === product.id ?
                          bid.price: product.price
```

The values emitted by these two observables are represented as the arguments `(product, bid)`, and this function returns either `product.price` or `bid.price`, depending on which observable emitted the value. This value will be used for rendering on the product-detail view.

Because the `combineLatest` operator requires both observables to emit a value to initially invoke the merge function, you apply the `startWith` operator (see http://mng.bz/OL9z) to ensure that there's an initial emission of the provided value before the observable will start making its regular emissions. For the initial values, you use the `product` for one observable, and either `BidMessage` or null for the other. When the `ProductDetailComponent` is initially rendered, the observable `bidService.priceUpdates$` emits null.

Your combined observable is declared in the `ngOnInit()` lifecycle hook, and its values are rendered in the template using the `async` pipe. You do it inside the `*ngIf` directive so the falsy values aren't rendered:

```
*ngIf="latestBids$ | async as price"
```

When the user clicks the bid button, you invoke `bidService.placeBid()`, which internally checks whether the connection to `BidServer` has to be opened or is already opened. The next listing from product-detail.component.html shows how the bid button is implemented in the template.

Listing 13.26 The bid button in the template

```
<button class="info__bid-button"
        mat-raised-button
        color="accent"
```

```
  (click)="placeBid(price + 5)">
    PLACE BID {{ (price + 5) | currency:'USD':'symbol':'.0' }}
</button>
```

Places the bid that's $5 higher than the latest bid/price

Shows the next bid amount on the button

Now let's see how the `BidService` class communicates with the server, using the WebSocket protocol.

USING RXJS TO COMMUNICATE WITH THE WEBSOCKET SERVER

In section 13.3.2, we showed a very basic way of writing the client code communicating with a WebSocket server. In ngAuction, you'll use a more robust WebSocket service included with RxJS, which means you can use it in any Angular project.

RxJS offers an implementation of the WebSocket service based on the `Subject` explained in section D.6 in appendix D. The RxJS `Subject` is both an observer and observable. In other words, it can receive and emit data, which makes it a good fit for handling WebSocket data streams. The RxJS `WebSocketSubject` is a wrapper around the standard browser `WebSocket` object and is located in the rxjs/websocket file.

> **TIP** Prior to RxJS 6, the `WebSocketSubject` class was located in the rxjs/observable/dom/WebSocketSubject file.

In its simplest form, `WebSocketSubject` can accept a string with the URL of the WebSocket endpoint or an instance of the `WebSocketSubjectConfig` object, where you can provide additional configuration. When your code subscribes to `WebSocketSubject`, it either uses the existing connection or creates a new one. Unsubscribing from `WebSocketSubject` closes the connection if there are no other subscribers listening to the same WebSocket endpoint.

When the server pushes data to the socket, `WebSocketSubject` emits the data as an observable value. In case of an error, `WebSocketSubject` emits the error like any other observable. If the server pushes data but there are no subscribers, the values will be buffered and emitted as soon as a new client subscribes.

> **NOTE** There's a difference in handling messages by a regular RxJS `Subject` and the `WebSocketSubject`, though. If you call `next()` on `Subject`, it emits data to all subscribers, but if you call `next()` on `WebSocketSubject`, it won't. Remember that there's a server between an observable and subscribers, and it's up to the server to decide when to emit values.

The ngAuction client that comes with this chapter includes the file shared/services/bid.service.ts, which uses `WebSocketSubject`. `BidService` is a singleton that's used only by `ProductDetailComponent`, which subscribes to it using the `async` pipe. When a user closes the product-detail view, the component gets destroyed, and the `async` pipe unsubscribes, closing the WebSocket connection. Let's review the code of script bid.service.ts.

Listing 13.27 bid.service.ts

```
import { WebSocketSubject } from 'rxjs/websocket';
...

export interface BidMessage {
  productId: number;
  price: number;
}

@Injectable()
export class BidService {
  private _wsSubject: WebSocketSubject<any>;
  private get wsSubject(): WebSocketSubject<any> {

    const closed = !this._wsSubject || this._wsSubject.closed;
     if (closed) {
      this._wsSubject = new WebSocketSubject(this.wsUrl);
     }
    return this._wsSubject;
  }

  get priceUpdates$(): Observable<BidMessage> {
    return this.wsSubject.asObservable();
   }

  constructor(@Inject(WS_URL) private readonly wsUrl: string) {}

  placeBid(productId: number, price: number): void {
    this.wsSubject.next(JSON.stringify({ productId, price }));
   }
}
```

A getter for the private property _wsSubject

The WebSocket-Subject was never created or is already disconnected.

Connects to BidServer

Gets a reference to the subject's observable

Injects the URL of the WebSocket server

Pushes the new bid to BidServer

The `BidService` singleton includes the `priceUpdates$` getter, which returns the observable. `ProductDetailComponent` uses this getter in `ngOnInit()`. That means `priceUpdates$` opens a WebSocket connection (through `this.wsSubject` getter) as soon as `ProductDetailComponent` is rendered, and the `async` pipe is a subscriber in the template of this component.

`BidService` also has a private property, `_wsSubject`, and the getter `wsSubject`, used internally. When the getter is accessed the very first time from `priceUpdates$`, the `_wsSubject` variable doesn't exist, and a new instance of `WebSocketSubject` is created, establishing a connection with the WebSocket server.

If a user navigates away from the product-detail view, the connection is closed. Because `BidService` is a singleton, if a user closes and reopens the product-detail view, the instance of `BidService` won't be re-created, but because the connection status is closed (`_wsSubject.closed`), it will be reestablished.

The URL of the WebSocket server (`WS_URL`) is stored in environment.ts for the dev environment and in environment.prod.ts for production. This value is injected into the `wsUrl` variable using the `@Inject` directive.

This concludes the code review of the ngAuction updates that implement communication between the Angular client and two servers. Run ngAuction as described in section 13.4.1, and ngAuction becomes operational.

Summary

- The WebSocket protocol offers unique features that aren't available with HTTP, which makes it a better choice for certain use cases. Both the client and the server can initiate communication.
- The WebSocket protocol doesn't use the request/response model.
- You can create an Angular service that turns WebSocket events into an observable stream.
- The RxJS library includes a `Subject`-based implementation of WebSocket support in the `WebSocketSubject` class, and you can use it in any Angular app.

14 Testing Angular applications

This chapter covers

- Using the Jasmine framework for unit testing
- Identifying the main artifacts from the Angular testing library
- Testing services, components, and the router
- Running unit tests against web browsers with the Karma test runner
- End-to-end testing with the Protractor framework

To ensure that your software has no bugs, you need to test it. Even if your application has no bugs today, it may have them tomorrow, after you modify the existing code or introduce new code. Even if you don't change the code in a particular module, it may stop working properly as a result of changes in another module or in the runtime environment. Your application code has to be retested regularly, and that process should be automated. You should prepare test scripts and start running them as early as possible in your development cycle.

This chapter covers two main types of testing for the frontend of web apps:

- *Unit testing*—Asserts that a small unit of code accepts the expected input data and returns the expected result. Unit testing is about testing isolated pieces of code, especially public interfaces.
- *End-to-end testing*— Asserts that the entire application works as end users expect and that all application parts properly interact with each other.

Unit tests are for testing the business logic of small, isolated units of code. They run reasonably fast, and you'll be running unit tests a lot more often than end-to-end tests. End-to-end (e2e) testing simulates user actions (such as button clicks) and checks that the application behaves as expected. During end-to-end testing, you shouldn't run unit-testing scripts.

> **NOTE** There are also integration tests that check that more that one app member can communicate. Whereas unit tests mock dependencies (for example, HTTP responses), integration tests use the real ones. To turn a unit test into an integration test, don't use mocks.

We'll start by covering the basics of unit testing with Jasmine, and then we'll show you how the Angular testing library is used with Jasmine. After that, you'll see how to use Protractor, the library for e2e tests. Toward the end of the chapter, we'll show you how to write and run e2e scripts to test the product-search workflow of ngAuction.

14.1 Unit testing

The authors of this book work as consultants on large projects for various clients. Pretty often these projects were written without unit tests in place. We're going to describe a typical situation that we've run into on multiple occasions.

A large app evolves over several years. Some of the developers who started writing the app are gone. A new developer joins the project and has to quickly learn the code and get up to speed.

A new business requirement comes in, and the new team member starts working on it. They implement this requirement in the existing function `doSomething()`, but the QA team opens another issue, reporting that the app is broken in a seemingly unrelated area. After additional research, it becomes obvious that the app is broken because of the code change made in `doSomething()`. The new developer doesn't know about a certain business condition and can't account for it.

This wouldn't have happened if unit (or e2e) tests were written with the original version of `doSomething()` and run as a part of each build. Besides, the original unit test would serve as documentation for `doSomething()`. Although writing unit tests seems like an additional, time-consuming task, it may save you a lot more time in the long run.

We like the definition given by Google engineer Elliotte Rusty Harold during one of his presentations—that a unit test should verify that a known, fixed input produces a known, fixed output. If you provide a fixed input for a function that internally uses

other dependencies, those dependencies should be mocked out, so a single unit test script tests an isolated unit of code.

Several frameworks have been created specifically for writing unit tests, and Angular documentation recommends Jasmine for this purpose (see the Angular documentation at http://mng.bz/0nv3). We'll start with a brief overview of Jasmine.

14.1.1 Getting to know Jasmine

Jasmine (https://jasmine.github.io/) enables you to implement a *behavior-driven development* (BDD) process, which suggests that tests of any unit of software should be specified in terms of the desired behavior of the unit. With BDD, you use natural language constructs to describe what you think your code should be doing. You write unit test specifications (specs) in the form of short sentences, such as "StarsComponent emits the rating change event."

Because it's so easy to understand the meaning of tests, they can serve as your program documentation. If other developers need to become familiar with your code, they can start by reading the code for the unit tests to understand your intentions. Using natural language to describe tests has another advantage: it's easy to reason about the test results, as shown in figure 14.1.

Figure 14.1 Running tests using Jasmine's test runner

> **TIP** Even though Jasmine comes with its own browser-based test runner, you'll be using a command-line-based test runner called Karma that can be easily integrated into the automated build process of your apps.

In BDD frameworks, a test is called a *spec*, and a combination of one or more specs is called a *suite*. A test suite is defined with the `describe()` function —this is where you describe what you're testing. Each spec in a suite is programmed as an `it()` function , which defines the expected behavior of the code under test and how to test it. The following listing shows an example.

Listing 14.1 A simple Jasmine test suite

```
describe('MyCalculator', () => {
   it('should know how to multiply', () => {
     // The code that tests multiplication goes here
  });
  it('should not divide by zero', () => {
     // The code that tests division by zero goes here
  });
});
```

A suite description and a function implementing the suite

A spec to test division

A spec to test multiplication

Testing frameworks have the notion of an *assertion*, which is a way of questioning whether an expression under test is true or false. If the assertion returns `false`, the framework throws an error. In Jasmine, assertions are specified using the `expect()` function, followed by *matchers*:`toBe()`, `toEqual()`, and so on. It's as if you're writing a sentence, "I expect 2 plus 2 to equal 4":

```
expect(2 + 2).toEqual(4);
```

Matchers implement a Boolean comparison between the actual and expected values. If the matcher returns `true`, the spec passes. If you expect a test result not to have a certain value, just add the keyword `not` before the matcher:

```
expect(2 + 2).not.toEqual(5);
```

> **NOTE** You can find the complete list of matchers in the type definition file @types/jasmine/index.d.ts, located in the directory node_modules. The Angular testing library adds more matchers, listed at http://mng.bz/hx5u.

In Angular, test suites have the same names as the files under test, adding the suffix .spec to the name. For example, the file application.spec.ts contains the test script for application.ts. Figure 14.2 shows a minimalistic test suite that can be located in the app.component.spec.ts file; it makes an *assertion* that the variable `app` is an instance of `AppComponent`. An assertion is the expectation plus the matcher.

Figure 14.2 shows a test suite containing a single spec. If you extract the texts from `describe()` and `it()` and put them together, you'll get a sentence that clearly

Figure 14.2 A minimalistic test suite

indicates what you're testing here: "ApplicationComponent is successfully instantiated." If other developers need to know what your spec tests, they can read the texts in `describe()` and `it()`. Each test should be self-descriptive so it can serve as program documentation.

> **TIP** Although the test shown in figure 14.2 was generated by Angular CLI, it's pretty useless because the chances that the `AppComponent` won't be successfully instantiated are close to zero.

The code in figure 14.2 instantiates `AppComponent` and expects the expression `app instanceof AppComponent` to evaluate to `true`. From the `import` statement, you can guess that this test script is located in the same directory as `AppComponent`.

> **NOTE** In Angular applications, you keep each test script in the same directory as the component (or service) under test, so if you need to reuse a component in another app, all related files are located together. If you use Angular CLI for generating a component or service, the boilerplate code for tests (the .spec.ts file) will be generated in the same directory.

If you want some code to be executed before each test (such as to prepare test dependencies), you can specify it in the *setup* functions `beforeAll()` and `beforeEach()`, which will run before the suite or each spec, respectively. If you want to execute some code right after the suite or each spec is finished, use the *teardown* functions `afterAll()` and `afterEach()`.

Let's see how to apply Jasmine API while unit-testing a TypeScript class.

14.1.2 Writing test scripts for a class

Imagine you have a `Counter` class with one `counter` property and two methods that allow incrementing or decrementing the value of this property.

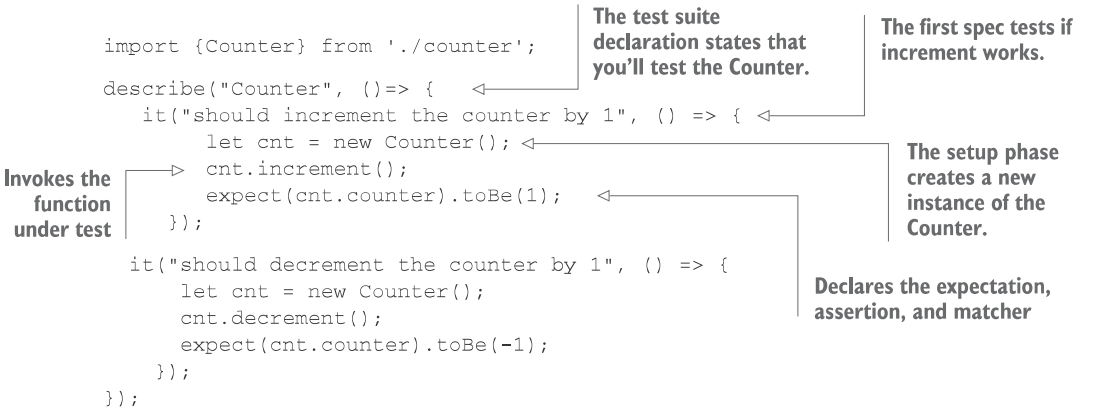
Listing 14.2 counter.ts

```
export class Counter {

  counter = 0;      <------ A class property

  increment() {     <------ A method to increment the value
     this.counter++;
  }

  decrement() {     <------ A method to decrement the value
     this.counter--;
  }
}
```

What do you want to unit-test here? You want to make sure that the `increment()` method increments the value of `counter` by one, and that the `decrement()` method decrements this value by one. Applying Jasmine terminology, you want to write a test suite with two specs.

Remember that a spec should test an isolated piece of functionality, so each spec should create an instance of the `Counter` class and invoke *only* *one* of its methods. The first version of the counter.spec.ts file is shown in the following listing.

Listing 14.3 counter.spec.ts

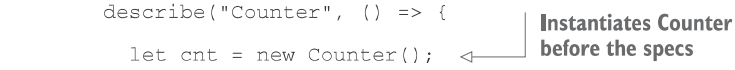



```
import {Counter} from './counter';

describe("Counter", ()=> {
   it("should increment the counter by 1", () => {
        let cnt = new Counter();
        cnt.increment();
        expect(cnt.counter).toBe(1);
    });

  it("should decrement the counter by 1", () => {
      let cnt = new Counter();
      cnt.decrement();
      expect(cnt.counter).toBe(-1);
    });
});
```

Each of your specs has similar functionality. The setup phase creates a fresh instance of the `Counter` class, then it invokes the method to be tested, and finally it declares the expectation with the `expect()` method . In one spec, you expect the `counter` to be `1`, and in another `-1`.

This suite of tests will work, but you have some code duplication here: each of the specs repeats the instantiation of `Counter`. In the refactored version of your test script, you'll remove the `Counter` instantiation from the specs and do it before the specs. Take a look at the new test version in the following listing. Is it correct?

Listing 14.4 Refactored counter.spec.ts




```
import {Counter} from './counter';

describe("Counter", () => {
  let cnt = new Counter();

  it("should increment the counter by 1", () => {
       cnt.increment();
       expect(cnt.counter).toBe(1);
    }
  );

  it("should decrement the counter by 1", () => {
      cnt.decrement();
      expect(cnt.counter).toBe(-1);
    }
  );
});
```

This test is not correct. Your test suite will create an instance of `Counter`, and the first spec will increase the `counter` value to `1` as expected. But when the second spec decrements the `counter`, its value becomes `0`, though the matcher expects it to be `-1`.

The final version of your test script, shown in the nest listing, fixes this mistake by creating the instance of `Counter` inside Jasmine's `beforeEach()` function.

Listing 14.5 The final version of counter.spec.ts

```
import {Counter} from './counter';

describe("Counter", () => {

  let cnt: Counter;

  beforeEach(() => cnt = new Counter()) ;

  it("should increment the counter by 1", () => {
       cnt.increment();
       expect(cnt.counter).toBe(1);
    }
  );

  it("should decrement the counter by 1", () => {
      cnt.decrement();
      expect(cnt.counter).toBe(-1);
    }
  );
});
```

Instantiates Counter inside beforeEach()

Now this script properly instructs Jasmine to create a new instance of `Counter` before running each spec of your suite. Let's see how to run it.

14.2 Running Jasmine scripts with Karma

For projects that don't use Angular CLI, you need to do lots of manual configurations to run Jasmine tests. Without Angular CLI, you start with installing Jasmine and its type definition files as follows:

```
npm install jasmine-core @types/jasmine --save-dev
```

Then you need to create a test.html file that includes script tags to load Jasmine and your specs (the TypeScript code needs to be precompiled into JavaScript). Finally, you need to manually load test.html in each browser you care about and watch whether your tests fail or pass.

But running unit tests from the command line is a better option, because that way you can integrate tests into the project build process. This is one of the main reasons for using a command-line test runner called Karma (see https://karma-runner.github.io). Along with that benefit, Karma has multiple useful plugins and can be used with many JavaScript testing libraries for testing against all major browsers.

Karma is used for testing JavaScript code written with or without frameworks. Karma can run tests to check whether your application works properly in multiple browsers (Chrome, Firefox, Internet Explorer, and so on). In non-Angular CLI projects, you can

install Karma and the plugins for Jasmine, Chrome, and Firefox, as shown in the following listing.

Listing 14.6 Installing Karma

```
npm install karma karma-jasmine --save-dev
npm install karma-chrome-launcher --save-dev
npm install karma-firefox-launcher --save-dev
```

Installs Karma and its Jasmine plugin

Installs the plugin to test in Chrome

Installs the plugin to test in Firefox

Then you need to prepare a configuration file, karma.conf.js, for your project—but you're spoiled by Angular CLI, which installs and configures everything you need for testing Angular apps, including Jasmine and Karma. We've generated a new project with Angular CLI and added the code described in the previous section to test the `Counter` class there. You'll find this project in the hello-jasmine directory. Figure 14.3 shows the structure of this project, marking all test-related files and directories.

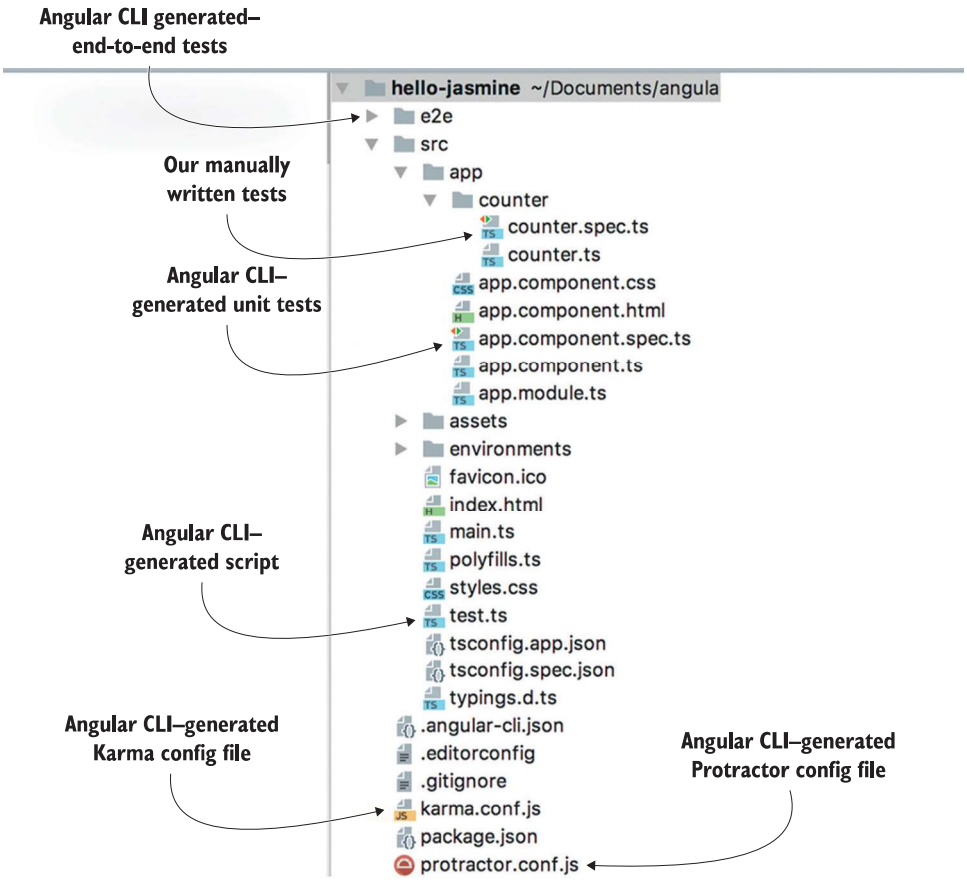


Figure 14.3 The hello-jasmine project

At the very top, you see the e2e directory, and at the bottom, the protractor.conf.js file, which were generated for end-to-end testing, described in section 14.4.

The counter.spec.ts file is the manually written test script described in the previous section. The app.component.spec.ts file was generated by Angular CLI for testing the `AppComponent`, and you'll see its content in section 14.3.1.

The generated file test.ts is the main testing script that loads all test scripts. The karma.conf.js file is used by the Karma runner as soon as you run the `ng test` command, which compiles and runs unit tests. After the tests are compiled, `ng test` uses the compiled script test.js to load the Angular testing library and all the .spec.ts files, and start the Karma runner. Figure 14.4. shows the output of the `ng test` command that in the hello-jasmine project.

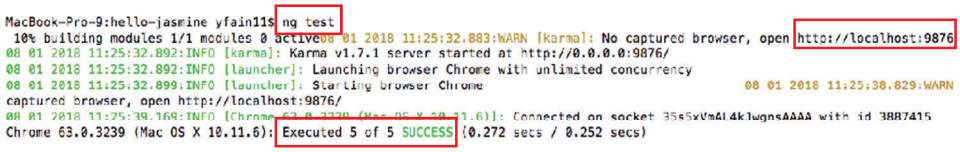


Figure 14.4 Running `ng test` in the hello-jasmine project

To run the tests, Karma starts the Chrome browser (the only one configured by Angular CLI) and runs five tests that end successfully. Why five? You wrote only two tests in the `counter.spec.ts` file, right? Angular CLI also generates the app.component .spec.ts file, which includes the test suite with three `it()` functions defined. Karma executes all files that have an extension .spec.ts.

> **NOTE** Angular CLI projects include the karma-jasmine-html-reporter package, and if you want to see the test results in the browser, open the URL http://localhost:9876.

You don't want to run tests from app.component.spec.ts at this point, so let's turn them off. If you want the test runner to skip some tests, rename their spec function from `it()` to `xit()`. Here, *x* is for *exclude*. If you want to skip the entire test suite, rename `describe()` to `xdescribe()`.

If you exclude the test suite in app.component.spec.ts, the tests will be automatically rerun, reporting that two tests ran successfully (those that you wrote for `Counter`), and three specs were skipped (those that were generated by Angular CLI):

```
Chrome 63.0.3239 (Mac OS X 10.11.6): Executed 2 of 5 (skipped 3)
 SUCCESS (0.03 secs / 0.002 secs)
```

As the number of specs grows, you may want to execute just some of them to see the results faster. Renaming a spec function from `it()` to `fit()` (*f* is for *force*) will execute only these tests while skipping the rest.

You know how to test, but why is still not clear

Let's say you know how to test the methods of your `Counter` class, but you still may have a million-dollar question: Why test such simple functions like `increment()` and `decrement()`? Isn't it obvious that they'll always work fine? In the real world, things change, and what used to be simple becomes not so simple anymore.

Say the business logic for the `decrement()` function changes, and the new requirement is not to allow `counter` to be less than `2`. The developer changes the `decrement()` code to look like this the following.

```
decrement(){
    this.counter >2 ? this.counter--: this.counter;
  }
```

Suddenly, you have two possible *execution paths*:

- The current counter value is greater than `2`.
- The current counter value is equal to `2`.

If you had the unit test for `decrement()`, the next time you run `ng test` it would fail, as follows:

```
Chrome 63.0.3239 (Mac OS X 10.11.6)
    ➥Counter should decrement the counter by 1
 FAILED
 Expected 0 to be -1.
     at Object.<anonymous> chapter14/hello-jasmine/
    ➥src/app/counter/counter.spec.ts:18:27)
    ...
```

The text describes the spec that failed.

The assertion failed because the code under test didn't decrement the counter that was equal to zero.

The fact that your unit test failed is a good thing, because it tells you that something changed in the application logic—in `decrement()`. Now the developer should see what changed and add another spec to the test suite so you have two `it()` blocks testing both execution paths of `decrement()` to ensure that it always works properly.

In the real world, business requirements change pretty often, and if developers implement them without providing unit tests for the new functionality, your app can become unreliable and will keep you (or production support engineers) awake at night.

TIP The output of the failed test may not be easy to read because it can include multiple lines of error stack trace. Consider using the continuous testing tool called Wallaby (see https://wallabyjs.com/docs), which shows you a short error message in your IDE right next to the code of the spec that failed.

NOTE In chapter 12, section 12.3.6, we explained how to automate the build process by running a sequence of npm scripts. If you add `ng test` to your build command, the build will be aborted if any of the unit tests fail. For

example, the build script can look like this: `"build": "ng test && ng build"`.

It's great that Angular CLI generates a Karma config file that works, but sometimes you may want to modify it based on your project needs.

14.2.1 Karma configuration file

When Angular CLI generates a new project, it includes karma.conf.js preconfigured to run Jasmine unit tests in the Chrome browser. You can read about all available configuration options at http://mng.bz/82cQ, but we'll just highlight some of them that you may want to modify in your projects. The generated karma.conf.js file is shown in the following listing.

Listing 14.7 Angular CLI-generated karma.conf.js file

```
module.exports = function (config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine', '@angular/cli'],
    plugins: [
      require('karma-jasmine'),
      require('karma-chrome-launcher')
       require('karma-jasmine-html-reporter'),
      require('karma-coverage-istanbul-reporter'),
       require('@angular/cli/plugins/karma')
     ],
    client:{
      clearContext: false // leave Jasmine Spec Runner
                          // output visible in browser
    },
    coverageIstanbulReporter: {
      reports: [ 'html', 'lcovonly' ],
      fixWebpackSourcePaths: true
    },
    angularCli: {
      environment: 'dev'
    },
    reporters: ['progress',
               'kjhtml'],
     port: 9876,
     colors: true,
    logLevel: config.LOG_INFO,
    autoWatch: true,
    browsers: ['Chrome'],
     singleRun: false
   });
};
```

Includes the plugin for testing in Chrome

Includes the code coverage reporter

Includes the Angular CLI plugin for Karma

Reports test progress on the console

Uses karma-jasmine-html-reporter

Runs the HTML reporter on this port

Lists the browsers to be used in tests

Runs in a watch mode

NOTE If you want Karma to print a message about each completed spec on the console, add karma-mocha-reporter as `devDependency` in package.json, add the line `require('karma-mocha-reporter')` to karma.conf.js, and replace the `progress` reporter with `mocha`. If you run tests in continuous integration (CI)

servers, use the karma-junit-reporter that can write test results into a file in JUnit XML format.

This configuration file uses only the Chrome plugin, but in real-world apps, you want to run tests in several browsers. The next section shows you how to add Firefox to the list of browsers to be used in tests.

Karma can report how well your code is covered with tests using the Istanbul reporter, and you can run the following command to generate the coverage report:

```
ng test --code-coverage
```

This will create a directory called coverage that will include an index.html file that loads the coverage report. For example, your hello-jasmine project includes one `AppComponent` and the `Counter` class, which are completely covered with unit tests. The generated report is shown in figure 14.5.

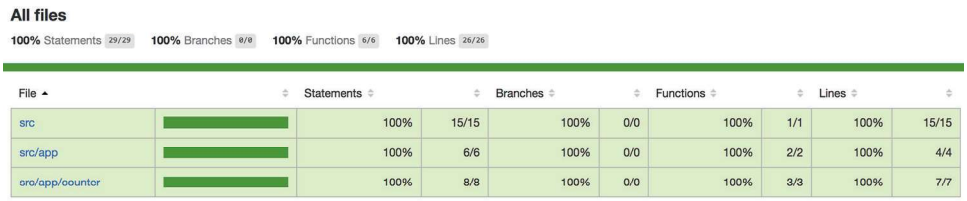


Figure 14.5 Test coverage report for the hello-jasmine project

NOTE Some organizations impose strict rules for code coverage, such as that at least 90% of the code must be covered with unit tests or the build must fail. To enforce such coverage, install the npm package karma-istanbul-threshold and add the `istanbulThresholdReporter` section to karma.conf.js. For more details, see http://mng.bz/544u.

14.2.2 Testing in multiple browsers

Typically, a developer doesn't manually test each and every code change in multiple browsers. Chrome is a preferred browser for dev mode, and you may be unpleasantly surprised when the tester reports that your app works well in Chrome, but produces errors in Safari, Firefox, or Internet Explorer. To eliminate these surprises, you should run unit tests in all browsers that matter to your users.

Luckily, that's pretty easy to set up with Karma. Let's say you want Karma to run tests not only in Chrome, but in Firefox as well (you have to have Firefox installed on your computer). First, install the karma-firefox-launcher plugin:

```
npm i karma-firefox-launcher --save-dev
```

Then, add the following line in the `plugins` section of karma.conf.js:

```
require('karma-firefox-launcher'),
```

Finally, add Firefox to the `browsers` list in karma.conf.js, so that it looks as follows:

```
browsers: ['Chrome', 'Firefox'],
```

> **TIP** If you need to set up a CI environment on a Linux server, you can either install Xvfb (a virtual display server) or use a *headless* browser (a browser without a UI). For example, you can specify `ChromeHeadless` to use the headless Chrome browser.

Now if you run the `ng test` command, it'll run the tests in both Chrome and Firefox. Install Karma plugins for each browser you care about, and this will eliminate surprises like "But it worked fine in Chrome!"

We've gone over the basics of writing and running unit tests. Let's see how to unit-test Angular components, services, and the router.

14.3 Using the Angular testing library

Angular comes with a testing library that includes the wrappers for some Jasmine functions and adds such functions as `inject()`, `async()`, `fakeAsync()`, and others.

To test Angular artifacts, you need to create and configure an Angular module for the class under test using the `configureTestingModule()` method of the `TestBed` utility, which allows you to declare modules, components, providers, and so on. For example, the syntax for configuring a testing module looks similar to configuring `@NgModule()`, as you can see in the following listing.

Listing 14.8 Configuring the testing module for your app

```
beforeEach(async(() => {                          <-- Runs this code asynchronously before each spec
   TestBed.configureTestingModule({               <-- Configures the testing module
     declarations: [
      AppComponent                                <-- Lists components under test
     ],
  }).compileComponents();                         <-- Compiles components
 }));
```

The `beforeEach()` function is used in test suites during the setup phase. With it you can specify the required modules, components, and providers that may be needed by each test. The `async()` function runs in the Zone and may be used with asynchronous code. The `async()` function doesn't complete until all of its asynchronous operations have been completed or the specified timeout has passed.

In an Angular app, the components are "magically" created and services are injected, but in test scripts, you'll need to explicitly instantiate components and invoke the `inject()` function or the `TestBed.get()` function to inject services. If a function under test invokes asynchronous functions, you should wrap such it into `async()` or `fakeAsync()`.

`async()` will run the function(s) under test in the Zone. If your test code uses timeouts, observables, or promises, wrap it into `async()` to ensure that the `expect()`

function is invoked after all the asynchronous functions are complete. If you don't do this, `expect()` may be executed before the results of async functions are in, and the test will fail. The `async()` function waits for async code to be finished, which is a good thing. On the other hand, such a wait may slow down the tests, and the `fakeAsync()` function allows you to eliminate the wait.

`fakeAsync()`identifies the timers in the code under test and replaces the code inside `setTimeout()`, `setInterval()`, or the `debounceTime()` with immediately executed functions as if they're synchronous, and executes them in order. It also gives you more-precise time control with the `tick()` and `flush()` functions, which allow you to fast-forward the time.

You can optionally provide the time value in milliseconds for fast-forwarding, so there's no need to wait, even if the async function uses `setTimeout()` or `Observable.interval()`. For example, if you have an input field that uses the RxJS operator `myInputField.valueChanges.debounceTime(500).subscribe()`, you can write `tick(499)` to fast-forward the time by 499 milliseconds and then assert that the subscriber didn't get the data entered in the input field.

You can use the `tick()` function only inside `fakeAsync()`. Calling `tick()` without the argument means that you want the code that follows to be executed after all pending asynchronous activities finish.

To see the tests from this section in action, open the unit-testing-samples project that comes with this chapter, run `npm install`, and then run `ng test`.

Let's see some of the APIs of the Angular testing library, starting with reviewing the code of the app.component.spec.ts file generated by Angular CLI.

14.3.1 Testing components

Components are classes with templates. If a component's class contains methods implementing some application logic, you can test them as you would any other function. But more often, you'll be testing the UI to see that the bindings work properly and that the component template displays expected data.

Under the hood, an Angular component consists of two parts: an instance of the class and the DOM element. Technically, when you write a unit test for a component, it's more of an integration test, because it has to check that the instance of the component class and the DOM object work in sync.

The Angular testing library offers the `TestBed.createComponent()` method,which returns a `ComponentFixture` object that gives you access to both the component and the native DOM object of the rendered template.

To access the component instance, you can use the `ComponentFixture.componentInstance` property, and to access the DOM element, use `ComponentFixture.nativeElement`. If you want to get access to the fixture's API (for example, to access the component's injector, run CSS query selectors, find styles or child nodes, or trigger an event handler), use its `DebugElement`, as in `ComponentFixture.debugElement.componentInstance` and `ComponentFixture.debugElement.nativeElement`, respectively. Figure 14.6 illustrates some of the properties of the `ComponentFixture` object, which also exist in `debugElement`.

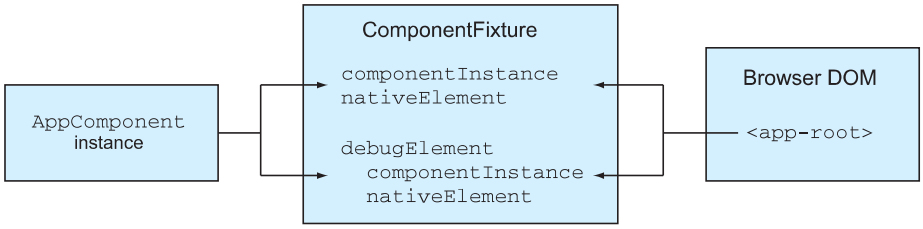


Figure 14.6 Properties of `ComponentFixture`

To update the bindings, you can trigger the change detection cycle on the component by invoking the `detectChanges()` method on the fixture. After change detection has updated the UI, you can run the `expect()` function to check the rendered values.

After configuring the test module, you usually perform the following steps to test a component:

1. Invoke `TestBed.createComponent()` to create a component.
2. Use a reference to `componentInstance` to invoke the component's methods.
3. Invoke `ComponentFixture.detectChanges()` to trigger change detection.
4. Use a reference to `nativeElement` to access the DOM object and check whether it has the expected value.

NOTE If you want change detection to be triggered automatically, you can configure the testing module with the provider for the `ComponentFixtureAutoDetect` service. Although this seems to be a better choice than manually invoking `detectChanges()`, this service only notices the asynchronous activities and won't react to synchronous updates of component properties.

Let's examine the code of the generated app.component.spec.ts file and see how it performs these steps. This Angular CLI–generated script declares a test suite containing three specs:

1. Check that the component instance is created.
2. Check that this component has a `title` property with the value `app`.
3. Check that the UI has an `<h1>` element with the text "Welcome to app!"

The code is shown in the following listing.

Listing 14.9 app.component.spec.ts

```
import { TestBed, async } from '@angular/core/testing';
import { AppComponent } from './app.component';

describe('AppComponent', () => {
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [
        AppComponent
      ],
```

Imports the required modules from the Angular testing library

Wraps component compilation into async()

In setup phase, configures the testing module asynchronously in the Zone

```
  }).compileComponents();
 }));
 it('should create the app', async(() => {
   const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.debugElement.componentInstance;
    expect(app).toBeTruthy();
 }));
 it(`should have as title 'app'`, async(() => {
   const fixture = TestBed.createComponent(AppComponent);
   const app = fixture.debugElement.componentInstance;
   expect(app.title).toEqual('app');
 }));
 it('should render title in a h1 tag', async(() => {
   const fixture = TestBed.createComponent(AppComponent);
   fixture.detectChanges();
    const compiled = fixture.debugElement.nativeElement;
    expect(compiled.querySelector('h1').textContent)
                      .toContain('Welcome to app!');
 }));
});
```

Compiles component(s) to inline styles and templates

Instantiates the AppComponent

Gets a reference to the component's instance

Checks that the coercion of the app to a Boolean yields true

Triggers change detection to update the component's DOM object

Gets a reference to the DOM object

Checks that the DOM object has the <h1> element containing this text

Note that the functions that instantiate the component are wrapped into `async()`. That's because a component can have a template and styles in separate files, and reading files is an asynchronous operation.

Invoking `detectChanges()` triggers change detection that updates the bindings on the DOM elements. After this is done, you can query the content of the DOM elements to assure that the UI shows the expected values.

NOTE Currently, Angular CLI generates the test with repeating `createComponent()` invocations. A better solution would be to write another `beforeEach()` function and create the fixture there.

Running `ng test` in a newly generated project will report that all tests passed. The browser opens at http://localhost:9876, and you'll see the testing report shown in figure 14.7.

Let's see what happens if you change the value of the `title` property in the `AppComponent` from `app` to `my app`. Because `ng test` runs in watch mode, the tests will automatically rerun, you'll see the messages about two failed specs on the console,

Figure 14.7 A successful run of `ng test`

and the list of specs will look like figure 14.8 (the failed specs are shown in red if you have the e-book).

The first failed spec message reads "AppComponent should have as title 'app'," and the second message is "AppComponent should render title in a h1 tag." These are the messages provided in the `it()` functions. Clicking any of the failed specs will open another page, providing more details and the stack trace.

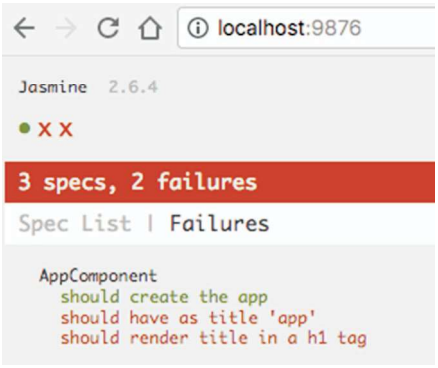


Figure 14.8 Spec list with failures

TIP Keep in mind that if your component uses lifecycle hooks, they won't be called automatically. You need to call them explicitly, as in `app.ngOnInit()`.

Let's add another spec in the next listing to ensure that if the `title` property changes in the `AppComponent` class, it'll be properly rendered in the UI.

Listing 14.10 A spec for the `title` update

```
it('should render updated title', async(() => {
  const fixture = TestBed.createComponent(AppComponent);
  const app = fixture.debugElement.componentInstance;
  app.title = 'updated app!';                    <-- Updates the title property
   fixture.detectChanges();                      <-- Forces change detection
   const compiled = fixture.debugElement.nativeElement;
  expect(compiled.querySelector('h1').textContent)
          .toContain('Welcome to updated app!'); <-- Checks that the UI reflects the updated title
 }));
```

Now `ng test` will run this extra spec and will report that it successfully finished. In this section, you used the generated test for `AppComponent`, but you'll see another script that tests a component in the hands-on section.

A typical component uses services for data manipulation, and you create mock services that return hardcoded (and the same) values to concentrate on testing the component's functionality. The specs for components should test only components; services should be tested separately.

14.3.2 Testing services

A service is a class with one or more methods, and you unit-test only the public ones, which in turn may invoke private methods. In Angular apps, you specify providers for services in `@Component` or `@NgModule`, so Angular can properly instantiate and inject

them. In test scripts, you also declare providers for services under test, but you do this inside `TestBed.configureTestingModule()` in the setup phase.

Also, if in Angular apps you can use the provider's token in the class constructor to inject a service, in tests, the injection is done differently. For example, you can explicitly invoke the `inject()` function. The other option to instantiate and inject a service is to use the `TestBed.get()` method, which uses the root injector, as shown in figure 14.9. This will work if the service provider is specified in the root testing module.

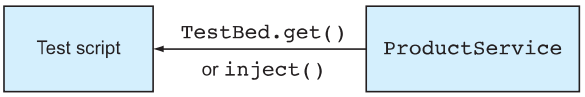


Figure 14.9 Injecting a service into the test script

Component-level injectors can be used as follows:

```
fixture.debugElement.injector.get(ProductService);
```

Let's generate a product service by running the following Angular CLI command:

```
ng g s product
```

This command will generate the files product.service.ts and product.service.spec.ts. The latter will contain the boilerplate code shown in the following listing.

Listing 14.11 product.service.spec.ts

```
import { TestBed, inject } from '@angular/core/testing';
import { ProductService } from './product.service';

describe('ProductService', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [ProductService]     <-- Configures the provider
    });
  });

  it('should be created',
     inject([ProductService],       <-- Injects the service
             (service: ProductService) => {
               expect(service).toBeTruthy();} <-- Implements the testing logic
      )
  );
});
```

If you needed to inject more than one service, the `inject()` function would list an array of DI tokens followed by the function with the argument list corresponding to the names of tokens:

```
inject([ProductService, OtherService],
    (prodService: ProductService, otherService: OtherService) => {...})
```

As you add methods to the `ProductService` class, you could test them similarly to testing methods in the `Counter` class, as you did earlier, but you need to consider a special case when a service relies on another service, such as on `HttpClient`. Making HTTP requests to a server during unit tests would slow them down. Besides, you don't want your unit tests to fail if the server's down. Remember, unit tests are for testing isolated pieces of code.

The code that comes with this chapter includes the unit-testing-samples project and the app called readfile. It includes `ProductService`, which uses `HttpClient` to read the data/products.json file, as shown in the following listing.

Listing 14.12 Reading data/products.json in a service

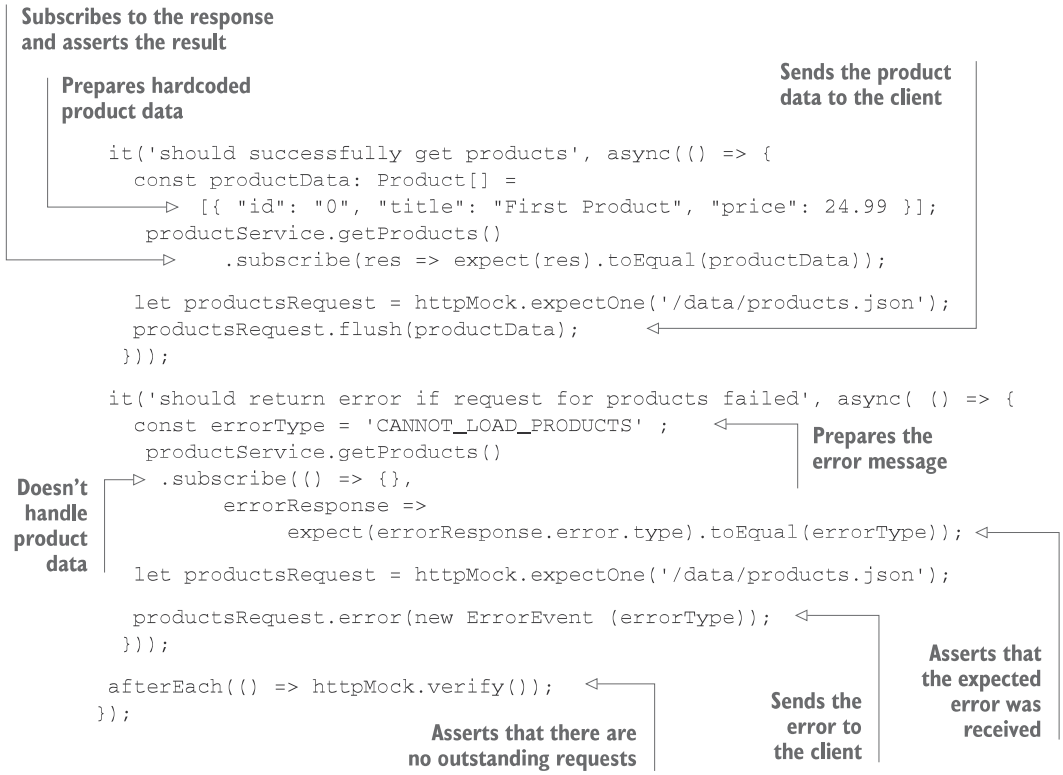
```
export class ProductService {

  constructor(private httpClient: HttpClient ) {}      <-- Injects HttpClient

  getProducts(): Observable<Product[]> {          <-- Reads the file
     return this.httpClient.get<Product[]>('/data/products.json');
  }
}
```

Let's write a unit test for the `getProducts()` method . You don't want your test to fail if someone removes the data/products.json file, because that wouldn't mean there's an issue in `getProducts()`. You'll mock the `HttpClient` with the help of `HttpTestingController` from `HttpClientTestingModule`. `HttpTestingController` doesn't make an HTTP request but allows you to emulate it using hardcoded data.

To add the hardcoded data to the response body, you'll use the `HttpTestingController.flush()` method, and to emulate an error, you'll use `HttpTestingController.error()`, as shown in the following listing.

Listing 14.13 product.service.spec.ts

```
import {TestBed, async} from '@angular/core/testing';
import {HttpClientTestingModule, HttpTestingController }
➥from '@angular/common/http/testing';
import {ProductService} from './product.service';
import {Product} from './product';

describe('ProductService', () => {
  let productService: ProductService;
  let httpMock: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],     <-- Includes HttpClientTestingModule to the testing module
       providers: [ProductService]
    });

    productService = TestBed.get(ProductService);     <-- Injects ProductService
     httpMock = TestBed.get(HttpTestingController);     <-- Injects HttpTestingController
   });
```



In the first spec, you hardcode the data for one product and then invoke `getProducts()` and subscribe to the response.

> **NOTE** Jasmine offers a `spyOn()` function that could intercept the specified function (for example, `getProducts()`), where you could just return a stub object with the expected data. But using such a spy wouldn't make an HTTP request. Because you use `HttpTestingController`, the HTTP request is made and will be intercepted by `HttpTestingController`, which won't be making a real HTTP request to read products.json but will take the hardcoded product data and send it through the HTTP machinery.

You expect the `getProducts()` method to make a single request to /data/products.json and return its mock, and this is what `expectOne()` is for. If no such request has been made, or if more than one such request has been made, the spec will fail.

With the real `HttpClient` service, invoking the `subscribe()` method would result in receiving either the data or an error, but with `HttpTestingController`, the subscriber won't get any data until you invoke `flush()` or `error()`. Here, you provide hardcoded data in the response body.

When Karma opens the browser with the test results, you can open Chrome Dev Tools in the Sources tab, find the source code for your spec file, and add breakpoints to debug your test code just as you'd do with any TypeScript code. In particular, if you

place a breakpoint in the line that invokes `flush()` in listing 14.13, you'll see that it's invoked before the code in `subscribe()`.

The `verify()` method tested all HTTP requests, and there are no outstanding ones. You assert this in the teardown phase after running each spec.

Note that the code in each spec is wrapped into the `async()` function. This ensures that your `expect()` calls will be made after all asynchronous calls from the spec are complete.

> **TIP** You can read about other techniques for replacing real services with mocks, stubs, and spies in the Angular testing documentation at https://angular.io/guide/testing.

Now let's see how to test the router.

14.3.3 Testing components that use routing

If a component includes routing, you may want to test different navigation functionality. For example, you may want to test that the router properly navigates where it's supposed to, that parameters are properly passed to the destination component, and that the guards don't let unauthorized users visit certain routes.

To test router-related functionality, Angular offers the `RouterTestingModule`, which intercepts navigation but doesn't load the destination component. For the test, you need the router configuration; you can either use the same one that's used in the application or create a separate configuration just for testing. The latter could be a better option if your route configuration includes many components.

A user can navigate the app either by interacting with the application or by entering a URL directly in the browser's address bar. The `Router` object is responsible for navigation implemented in your app code, and the `Location` object represents the URL in the address bar. These two objects work in sync.

To test if the router properly navigates your app, invoke `navigate()` and `navigateByUrl()` in your specs, and pass parameters, if needed. The `navigate()` method takes an array of routes and parameters as an argument, whereas `navigateByUrl()` takes a string representing the segment of the URL you want to navigate to.

If you use `navigate()`, you specify the configured path and route params, if any. If the router is properly configured, it should update the URL in the address bar of the browser. To illustrate this, you'll reuse the code of one of the apps from chapter 3, but you'll add the spec file. In that app, the router configuration for the `AppComponent` includes the path /product/:id, as shown in the following listing.

Listing 14.14 A fragment from app.routing.ts

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

A default route

A route with a parameter

When the user clicks the Product Details link, the app navigates to the `ProductDetailComponent`, as shown in the following listing.

Listing 14.15 app.component.ts

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

A link to navigate to the product-detail view

The value to be passed to the product-detail view

In the app.component.spec.ts file, you'll test that when the user clicks the Product Details link, the URL includes the segment /product/1234. The `Router` and `Location` objects will be injected by using the `TestBed.get()` API. To emulate the click on the Product Details link, you need to get access to the corresponding DOM object, which you do by using the `By.css()` API. The utility class `By` has the `css()` method, which matches elements using the provided CSS selector. Because your app component has two links, you assign `id=product` to the product-details link so you can get ahold of it by invoking `By.css('#product')`.

To emulate the click on the link, you use the `triggerEventHandler()` method with two arguments. The first argument has the value `click` that represents the click event. The second argument has the value `{button: 0}` that represents the event object. The `RouterLink` directive expects the value to include the property `button` with the number that represents the mouse button, and zero is for the left mouse button, as shown in the following listing.

Listing 14.16 app.component.spec.ts

```
// imports omitted for brevity
describe('AppComponent', () => {
  let fixture;
  let router: Router;
  let location: Location;

    beforeEach(async(() => {
      TestBed.configureTestingModule({
        imports: [RouterTestingModule.withRoutes(routes)],
         declarations: [
          AppComponent, ProductDetailComponent, HomeComponent
        ]}).compileComponents();
    }));

  beforeEach(fakeAsync(() => {
    router = TestBed.get(Router);
```

Loads the routes configuration

Injects the Router object

```
    location = TestBed.get(Location);
    fixture = TestBed.createComponent(AppComponent);
   router.navigateByUrl('/');
    tick();
    fixture.detectChanges();
   }));

  it('can navigate and pass params to the product detail view',
   fakeAsync(() => {
    const productLink = fixture.debugElement.query(By.css('#product'));
    productLink.triggerEventHandler('click', {button: 0});
    tick();
    fixture.detectChanges();
    expect(location.path()).toEqual('/product/1234');
   }));
});
```

Injects the Location object

Triggers change detection

Get access to the product-details link

Clicks the link

Checks the assertion

The `fakeAsync()` function wraps the navigation code (the asynchronous operation), and the `tick()` function ensures that the asynchronous navigation finishes before you run the assertion.

Figure 14.10 shows the sequence of actions performed by the preceding script.

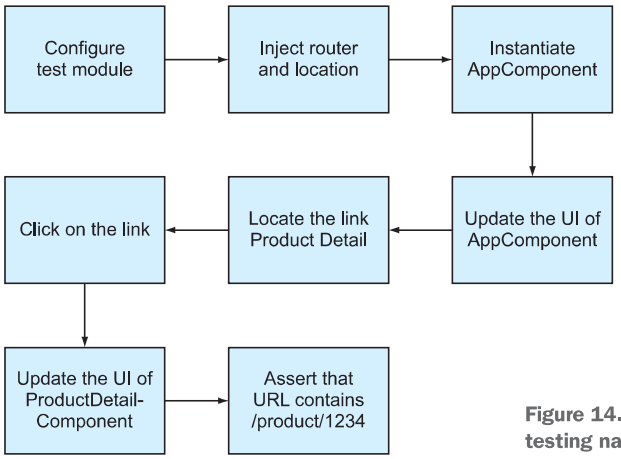


Figure 14.10 Steps in testing navigation

The `ng test` command will run all unit tests in the unit-testing-samples project, which has three apps. All eight specs should successfully complete. The eighth spec will report "AppComponent can navigate and pass params to the product detail view."

> **NOTE** To make unit testing a part of your automated build process, integrate the `ng test` command into the build process by adding `&& ng test` to the npm build script described in section 12.3.6 in chapter 12.

Unit testing the functionality implemented in the route guards is another practical use case. In chapter 4, we covered such guards as `CanActivate`, `CanDeactivate`, and

`Resolve`. Because guards are services, you can test them separately, as explained in the preceding section.

This concludes our coverage of unit-testing basics. Unit tests assert that each artifact of your Angular app works as expected in isolation. But how can you ensure that several components, services, and other artifacts play well together without the need to manually test each workflow?

14.4 End-to-end testing with Protractor

End-to-end (E2E) testing is for testing the entire app workflow by simulating user interaction with the app. For example, the process of placing an order may use multiple components and services. You can create an E2E test to ensure that this workflow behaves as expected. Also, if in unit tests you're mocking dependencies, E2E tests will use the real ones.

To manually test a specific workflow like the login functionality, a QA engineer prepares an ID/password that works, opens the login page, enters the ID/password, and clicks the Login button. After that, QA wants to assert that the landing page of your app is successfully rendered. The tester can also run another test to ensure that if the wrong ID/password is entered, the landing page won't be rendered. This is a manual way of E2E testing of the login workflow.

Protractor is a testing library that allows you to test app workflows *simulating* user actions without the need to perform them manually. You still need to prepare test data and script the test logic, but the tests will run without human interaction.

By default, Protractor uses the Jasmine syntax for tests, unless you manually configure another supported framework (see http://mng.bz/d64d). So your E2E test scripts will use already familiar `describe()` and `it()` blocks plus an additional API.

14.4.1 Protractor basics

While manually testing app workflows, a user "drives" the web browser by entering data, selecting options, and clicking buttons. Protractor is based on Selenium WebDriver (see http://www.seleniumhq.org/docs/03_webdriver.jsp) that can automatically drive the browser, based on the provided scripts. Protractor also includes an Angular-specific API for locating UI elements.

In your setup, Protractor will run the web browser and tests on the same machine, so you need Selenium WebDriver for the browser(s) you want to run the tests in. The other option would be to set up a separate machine for testing and run Selenium Server there. Selenium offers implementations of WebDriver for different programming languages, and Protractor uses the one called WebDriverJS.

When you generate a new project with Angular CLI, it includes Protractor and its configuration files as well as the e2e directory with sample test scripts. Prior to Angular 6, the e2e directory included three files, as shown in figure 14.11. Starting from Angular 6, the generated e2e directory includes the configuration file protractor .conf.js.

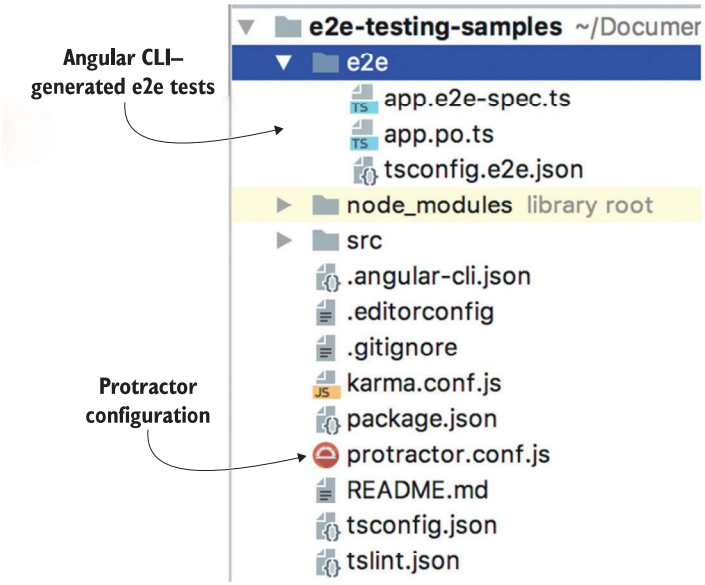


Figure 14.11 Angular CLI-generated E2E code

> **TIP** Starting from Angular CLI 6, when you generate a new project, it includes two apps: one is a project for your app, and another app contains the basic E2E tests.

You run the E2E tests by entering the `ng e2e` command , which loads the test scripts based on the configuration in the protractor.conf.js file. That file by default assumes that all E2E test scripts are located in the e2e directory, and the app has to be launched in Chrome.

Listing 14.17 A fragment from protractor.conf.js

```
specs: [
  './e2e/**/*.e2e-spec.ts' <-- Where the test scripts are
 ],
capabilities: {
  'browserName': 'chrome' <-- Which browser to run your app in
 },
directConnect: true  <-- Connects to the browser directly without the server
```

The `ng e2e` command builds the app bundles, starts the Node instance, and loads the test scripts, Protractor, and Selenium WebDriver. Protractor launches your app in the browser(s), and your test scripts communicate with the browser using the API of Protractor and WebDriverJS. Figure 14.12 shows the E2E test players used in this chapter's examples.

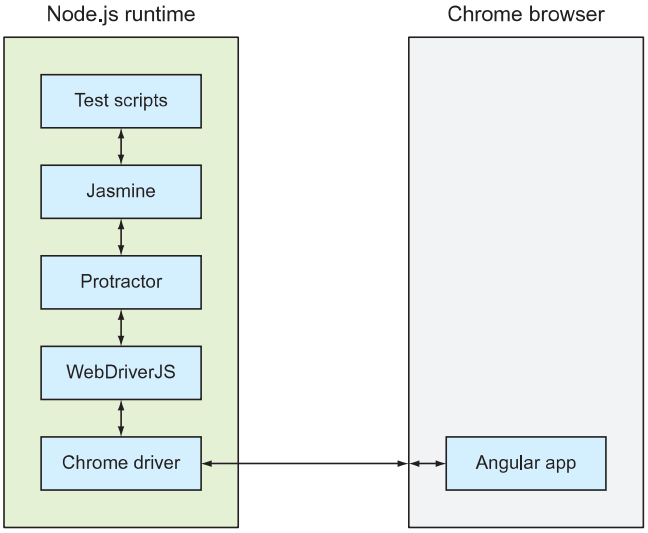


Figure 14.12 Angular CLI–generated E2E code

Prior to running your test scripts, Protractor unzips the browser-specific driver (for example, ChromeDriver) into the node_modules/webdriver-manager/selenium folder so Selenium WebDriver can properly communicate with the browser. During the tests, Protractor will launch the browser, and after the tests finish, Protractor will close it.

Protractor can use scripts created in different unit-testing frameworks (Jasmine is a default one), and each of them may have a different API for locating and representing page elements. To spare you from changing E2E scripts if you decide to switch to another unit-testing framework, Protractor comes with an API (see www.protractortest.org/#/api) that works with all supported frameworks:

- `browser` provides an API to control the browser, for example `getCurrentUrl()`, `wait()`, and so on.
- `by` is a locator for finding elements in Angular applications by ID, CSS, button or link text, and so forth.
- `element` offers an API for finding and working with a single element on a web page.
- `element.all` is used for finding and working with collections of elements, for example, iterating over the elements of an HTML list or table.

TIP `$("selector")` is an alias for `element(by.css("selector"))`, and `$$("selector")` is an alias for `element.all(by.css("selector"))`.

Although in Angular apps you can use the structural directive `*ngFor` for rendering a collection of UI elements, in tests you should use `element.all` for referring to and finding elements in a collection.

TIP Though Protractor defines its own API, it also exposes the WebDriver API, as in `browser.takeScreenshot()`.

The E2E tests load the real app in the browser, locate elements on the page, and can programmatically click buttons and links, fill out forms with data, submit them to the server, and then again locate the elements on the resulting page to ensure that they have the expected content. You can write an E2E test using one of the following approaches:

- In the same script, locate DOM elements by using their IDs or CSS classes and assert that the application logic works correctly. The IDs or CSS classes may change over time, so if you have several scripts testing the same page, you need to update each script accordingly.
- Implement the *Page Object* design pattern (see https://martinfowler.com/bliki/PageObject.html) by writing the expectations and assertions in one file, and in another, write the code that interacts with the UI elements and invokes the app's API. The page object can implement UI interaction with either the entire page or its part (for example, the toolbar), and can be reused by multiple tests. Should the CSS of the HTML elements change, you'll need to modify a single page object script.

Tests written using the first approach are difficult to read because they don't provide an easy way of understanding which workflows are implemented on the page. You'll use the second approach, where all UI interactions are implemented in the page objects (.po.ts files), and the specs with assertions are in the scripts (.e2e-spec.ts files). This approach reduces code duplication because you don't need to copy-paste the element locators if multiple specs need to access the same HTML element. A page object can serve as a single place for simulating user activity for important workflows, such as `login()` or `getProducts()`, rather than having these activities scattered throughout the tests.

Let's look at the E2E test generated by Angular CLI for the new projects.

14.4.2 Angular CLI–generated tests

When you generate a new project with Angular CLI, it creates a directory, e2e, that contains three files:

- *app.po.ts*—The page object for `AppComponent`
- *app.e2e-spec.ts*—The E2E test for the generated `AppComponent`
- *tsconfig.e2e.json*—The TypeScript compiler options

The app.po.ts file contains a simple `AppPage` class with just two methods, as shown in listing 14.18. The first one contains the code to navigate to the root page of the component, and the second has code to locate the HTML element by CSS and get its text. This page object is the only place that contains code locating elements by CSS.

Listing 14.18 The generated app.po.ts file

```
import {browser, by, element} from 'protractor';

export class AppPage {
  navigateTo() {
    return browser.get('/'); <-- Navigates to the default route
   }

  getParagraphText() {
    return element(by.css('app-root h1')).getText(); <-- Gets the text from the <h1> element
   }
}
```

The code for the app.e2e-spec.ts file is shown in listing 14.19. This test looks very similar to the unit tests shown in the last section. Note that this file doesn't include the code that directly interacts with the HTML page; it uses the API of the page object instead.

Listing 14.19 The generated app.e2e-spec.ts file

```
import {AppPage} from './app.po';

describe('e2e-testing-samples App', () => {
  let page: AppPage;

  beforeEach(() => {
    page = new AppPage(); <-- Creates an instance of the page
   });

  it('should display welcome message', () => {
    page.navigateTo(); <-- Navigates to the default route
     expect(page.getParagraphText()).toEqual('Welcome to app!'); <-- Asserts that the text returned by getParagraphText() is correct
   });
});
```

Because app.e2e-spec.ts doesn't contain any element locators, it's easy to follow the test logic: you navigate to the landing page and retrieve the content of a paragraph. You can run the preceding E2E test using the command `ng e2e`.

> **NOTE** E2E tests run slower that unit tests, and you don't want to run them each time you save a file, as you did with `ng test` in the last section. Besides, instead of creating E2E tests for each and every workflow, you may want to identify the most important ones and run tests just for them.

Now that you've seen how generated tests work, you can write your own E2E test.

14.4.3 Testing a login page

The E2E test from the preceding section didn't include a workflow that would require data entry and navigation. In this section, you'll write a test for an app that uses a form and routing. The code that comes with this chapter includes a project called

e2e-testing-samples with a simple app that has a login page and a home page. The routes in this app are configured in the following listing.

Listing 14.20 Route configurations

```
[{path: '', redirectTo: 'login', pathMatch: 'full'},
  {path: 'login', component: LoginComponent},
  {path: 'home', component: HomeComponent}]
```

Redirects the base URL to the login page

Renders the login component

Renders the home component

The template of the `HomeComponent` has just one line:

```
<h1>Home Component</h1>
```

The login component in the following listing has a Login button and a form with two fields for entering ID and password. If a user enters `Joe` as the ID and `password` as the password, your app navigates to the home page; otherwise, it stays on the login page and shows the message "Invalid ID or password."

Listing 14.21 login.component.ts

```
@Component({
  selector: 'app-home',
  template: `<h1 class="home">Login Component</h1>
  <form #f="ngForm" (ngSubmit)="login(f.value)">
     ID: <input name="id" ngModel/><br>
    PWD: <input type="password" name="pwd" ngModel=""/><br>
    <button type="submit">Login</button>
    <span id="errMessage"
          *ngIf="wrongCredentials">Invalid ID or password</span>
   </form>
  `
})
export class LoginComponent {
  wrongCredentials = false;
  constructor(private router: Router) {}

  login(formValue) {
    if ('Joe' === formValue.id && 'password' === formValue.pwd) {
      this.router.navigate(['/home']);
       this.wrongCredentials = false;
    } else {
      this.router.navigate(['/login']);
       this.wrongCredentials = true;
    }
  }
}
```

A login form

The invalid login message

Router injection

Navigation to the home page

Navigation to the login page

Your tests are located in the e2e directory and include two page objects, login.po.ts and home.po.ts, and one spec, login.e2e-spec.ts. The page object for the home page contains a method to return the header's text. The following listing shows home.po.ts.

Listing 14.22 home.po.ts

```
import {by, element} from 'protractor';

export class HomePage {
  getHeaderText() {
    return element(by.css('h1')).getText();
  }
}
```

The login page object uses locators to get references to the form fields and the button. The `login()` method simulates user actions: entering the ID and password and clicking the Login button. The `navigateToLogin()` method instructs the browser to visit the URL configured to the login component—for example, http://localhost:4200/login. The `getErrorMessage()` method returns the login error message that may or may not be present on the page. login.po.ts is shown in the following listing.

Listing 14.23 login.po.ts

```
import {browser, by, element, $} from 'protractor';

export class LoginPage {

  id = $('input[name="id"]');
   pwd = $('input[name="pwd"]');
   submit = element(by.buttonText('Login'));
   errMessage = element(by.id('errMessage'));

  login(id: string, password: string): void {
    this.id.sendKeys(id);
     this.pwd.sendKeys(password);
     this.submit.click();
   }

  navigateToLogin() {
    return browser.get('/login');
   }

  getErrorMessage() {
    return this.errMessage;
   }
}
```

Locates the page elements using $ as an alias for element(by.css())

Enters the provided ID and password

Clicks the Login button

Navigates to the login page

Returns the login error message

This page object makes the login procedure easy to understand. The `sendKey()` method is used for simulating data entry, and `click()` simulates a button click.

Now let's review the test suite for the login workflow. It instantiates the login page object and includes two specs: one for testing a successful login and another for a failed one.

The first spec instructs Protractor to navigate to the login page and log in the user with the hardcoded data `Joe` and `password`. If the login was successful, the app navigates to the home page, and you assert this by checking that the URL in the browser

contains /home. You also assert that the rendered page contains the header "Home Component."

The spec for the failed login asserts that the app stays on the login page and the error message is displayed. Note in the following listing that this script has no code directly interacting with the UI.

Listing 14.24 login.e2e-spec.ts

```
import {LoginPage} from './login.po';
import {HomePage} from './home.po';
import {browser} from 'protractor';

describe('Login page', () => {
  let loginPage: LoginPage;
  let homePage: HomePage;

  beforeEach(() => {
    loginPage = new LoginPage();
  });

  it('should navigate to login page and log in', () => {
     loginPage.navigateToLogin();
     loginPage.login('Joe', 'password');

    const url = browser.getCurrentUrl();
     expect(url).toContain('/home');

     homePage = new HomePage();
     expect(homePage.getHeaderText()).toEqual('Home Component');
   });

  it('should stay on login page if wrong credentials entered',
     () => {
     loginPage.navigateToLogin();
    loginPage.login('Joe', 'wrongpassword');

    const url = browser.getCurrentUrl();
    expect(url).toContain('/login');
     expect(loginPage.getErrorMessage().isPresent()).toBe(true);
   });
});
```

Instantiates the login page object

A spec for a successful login

Navigates to the login page

Logs in with proper credentials

Gets the browser's URL

Asserts that the URL contains /home

Instantiates the home page object

Asserts that the page header is correct

A spec for a failed login

Executes the failed login

Asserts that the error message is shown

Asserts that the app still shows the login page

The `LoginComponent` uses the `*ngIf` structural directive to conditionally show or hide the login error message, and your failed login spec asserts that the error message is present on the page.

Sometimes you need to wait for certain operations to complete before making assertions. For example, the `login()` method in your page object ends with the button click, and the spec for the successful login contains the assertion that the URL contains /home.

This assertion will always be true because your login process completes in no time as it doesn't connect to an authentication server to check user credentials. In the real world, the authentication could take a couple of seconds, and the assertion for /home could run sooner than the URL changes to /home, causing the test to fail.

In such cases, you can invoke the `browser.wait()` command , where you can specify the condition to wait for. In the hands-on section, you'll write a test that clicks the Search button that makes an HTTP request for products, which needs some time to finish. There, you'll use a helper function that waits for the URL to change before making assertions.

Run this test with the `ng e2e` command, and you'll see how Protractor opens the Chrome browser for a short time, fills out the form, and clicks the Login button. The Terminal window shows the output, which you can see in figure 14.13.

```
[14:32:09] I/update - chromedriver: file exists /Users/yfain11/Documents/angular
typescript/code-samples/chapter14/e2e-testing-samples/node_modules/webdriver-man
ager/selenium/chromedriver_2.35.zip
[14:32:09] I/update - chromedriver: unzipping chromedriver_2.35.zip
[14:32:09] I/update - chromedriver: setting permissions to 0755 for /Users/yfain
11/Documents/angulartypescript/code-samples/chapter14/e2e-testing-samples/node_m
odules/webdriver-manager/selenium/chromedriver_2.35
[14:32:09] I/update - chromedriver: chromedriver_2.35 up to date
[14:32:10] I/launcher - Running 1 instances of WebDriver
[14:32:10] I/direct - Using ChromeDriver directly...
Jasmine started

  Login page
    ✓ should navigate to login page and log in
    ✓ should stay on login page if wrong credentials entered

Executed 2 of 2 specs SUCCESS in 2 secs.
[14:32:13] I/launcher - 0 instance(s) of WebDriver still running
[14:32:13] I/launcher - chrome #01 passed
```

Figure 14.13 Running E2E tests for the login app

Both specs from your E2E test passed. If you want to see the tests fail, remove the `<h1>` tags in the template of the `HomeComponent` or modify the valid credentials to anything other than `Joe` and `password` in the `LoginComponent`. Changing the names of the form fields in the template of `LoginComponent` will also cause the test to fail because the WebDriver locators won't find these elements on the login page.

Using async and await in E2E tests

Protractor uses WebDriverJS. Its API is entirely asynchronous, and its functions return promises. All asynchronous operations (for example, `sendKey()` and `click()`) are placed in the queue of pending promises called *control-flow queue* using the WebDriver promise manager to ensure that assertions (such as `expect()` functions) run after asynchronous operations.

(continued)

Because the WebDriver promise manager doesn't execute async functions right away but places them in a queue instead, it's hard to debug this code. That's why WebDriver's promise manager is being deprecated, and you can use the `async` and `await` keywords to ensure that flow is properly synchronized (see http://mng.bz/f72u for details).

For example, the following code declares a `login()` method.

```
async login(id: string, password: string) {
   await this.id.sendKeys(id);
   await this.pwd.sendKeys(password);
   await this.submit.click();
 }
```

Declares that the function returns a promise

Waits for the promise to be resolved or rejected

You can't use the `async/await` keywords with WebDriver's promise manager, so you need to turn if off by adding the following option in protractor.conf.js:

```
SELENIUM_PROMISE_MANAGER: false
```

This chapter has enough material to get you started with unit and E2E testing of Angular apps. Both Jasmine and (especially) Protractor offer more APIs that can be used in tests. For more detailed coverage, check out the book *Testing Angular Applications* (Jesse Palmer et al., Manning, 2018), with details at www.manning.com/books/testing-angular-applications.

If after getting familiar with the combination of Protractor and the Selenium ecosystem you'd like to find a simpler solution for E2E testing of your apps, take a look at the Cypress framework available at https://www.cypress.io. It's a new but very promising kid on the block. Meanwhile, let's add some Protractor E2E tests to ngAuction.

14.5 Hands-on: Adding an E2E test to ngAuction

The goal of this exercise is to add one E2E test to the ngAuction app, which you can find in the ng-auction folder in the source code that comes with this chapter. We took the ngAuction project from chapter 13 and added to it the E2E test for the product-search workflow. This test will use the price range from $10 to $100 to assert that matching products are retrieved from the server and rendered in the browser.

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

Prior to running this E2E test, you need to run `npm install` in the server directory, compile the code with the `tsc` command, and start the server by running the following command:

```
node build/main
```

Now you're ready to review and run the tests located in the client directory of ngAuction.

14.5.1 E2E testing of the product-search workflow

To perform product search, a real user would need to fulfill the following steps:

1. Open the landing page of ngAuction.
2. Click the Search button in the top-left corner so the search panel will show up.
3. Enter search criteria for products.
4. Click the Search button to see the search results.
5. Browse the products that meet the search criteria.

Your E2E test will consist of two files located in the e2e directory: the page object in the search.po.ts file and the test suite in search.e2e-spec.ts. All assertions will be programmed in the search.e2e-spec.ts file, but the page object will implement the following logical steps:

1. Find the Search button and click it.
2. Fill out the search form with data.
3. Click the Search button.
4. Wait until the server returns and renders products in the browser.
5. Check to see that the browser rendered products.

To ensure that your search will return some products, your test will use a wide range of prices from $10 to $100 as the search criteria.

In several cases, you'll be checking that the browser URL is what you expect it to be, so we'll remind you how the routes are configured in the home.module.ts in ngAuction, as shown in the following listing.

Listing 14.25 Routes configuration from the home module

```
[
  {path: '', pathMatch: 'full', redirectTo: 'categories'},
  {path: 'search', component: SearchComponent},
  {path: 'categories',
    children: [
      { path: '', pathMatch: 'full', redirectTo: 'all'},
      { path: ':category', component: CategoriesComponent},
    ]
  }
]
```

Let's start by identifying the HTML elements that will participate in our test. The file app.component.html includes the markup in the following listing for the Search button.

Listing 14.26 The Search button on the toolbar

```
<button mat-icon-button
        id="search"
         class="toolbar__icon-button"
        (click)="sidenav.toggle()">
  <mat-icon>search</mat-icon>
</button>
```

The added ID simplifies the code for locating this button.

Your page object will contain the lines in the following listing to locate the button and click it.

Listing 14.27 The beginning of the `SearchPage` class

```
export class SearchPage {

  performSearch(minimalPrice: number, maximumPrice: number) {
     const searchOnToolbar = element(by.id('search'));
     searchOnToolbar.click();
    ...
  }
}
```

A method for searching products by price range

Locating the Search button

The button click to display the search form

After the button is clicked, the search form is displayed, and you locate the fields for the minimum and maximum prices and fill them with the provided prices, as shown in the following listing.

Listing 14.28 Entering the search criteria

```
const minPrice = $('input[formControlName="minPrice"]');
 const maxPrice = $('input[formControlName="maxPrice"]');
 minPrice.sendKeys(minimalPrice);
 maxPrice.sendKeys(maximumPrice);
```

Locates the form fields for prices

Fills out some of the form fields

If the user did this manually, the search form would look like figure 14.14.

Now that the search criteria is entered, you need to locate and click the form's Search button to perform the product search. If you run ngAuction and enter the min and max prices as $10 and $100, and then click the Search button, the resulting view will show the products, and the browser URL will look like this: http:// localhost:4200/ search?minPrice=10&maxPrice=100.

But it'll take a second before the HTTP request is complete and the URL changes. The real user would be patiently waiting until the search results appeared. But in your test script, if you try to assert that the URL contains the search segment right after the button click, the assertion may or may not be true depending on how fast your server responds.

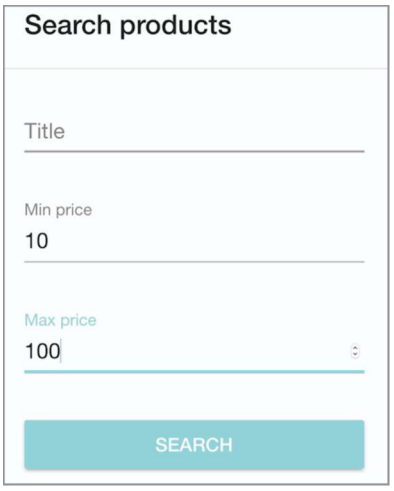


Figure 14.14 The form with search criteria

You didn't need to worry about delays in login.po.ts from section 14.4.3, because no server requests were made there, and the URL changed instantaneously. This time, you want to wait until the URL changes before returning from the method `performSearch()`.

You'll use the `ExpectedConditions` class , where you can define the condition to wait for. Then, by invoking `browser.wait()`, you can wait for the expected condition to become true—otherwise, the test has to fail by timeout. The following code listing locates and clicks the Search button and then waits until the URL changes to contain the /search segment.

Listing 14.29 Clicking the form's Search button

```
const searchOnForm = element(by.buttonText('SEARCH'));
 searchOnForm.click();
 const EC = protractor.ExpectedConditions;
 const urlChanged = EC.urlContains('/search');
 browser.wait(urlChanged, 5000,
              'The URL should contain /search');
```

Locates the Search button

Clicks the Search button

Declares the constant for the expected condition

Waits for the expected condition for up to 5 seconds or fails

The message to display in case of timeout

Uses the urlContains() API to check the expected condition

This code waits for up to 5 seconds, and if the URL doesn't change, it fails, printing the message shown in figure 14.15. You may need to increase the timeout value depending on how fast the product search is performed on your computer.

```
1) ngAuction search should perform the search for products that cost from $10 to $100
  - Failed: The URL should contain /search
  Wait timed out after 5001ms

Executed 1 of 1 spec (1 FAILED) in 4 secs.
```

Figure 14.15 The test fails on timeout.

If the user manually searched for products in the price range between $10 and $100, the resulting view could look like figure 14.16.

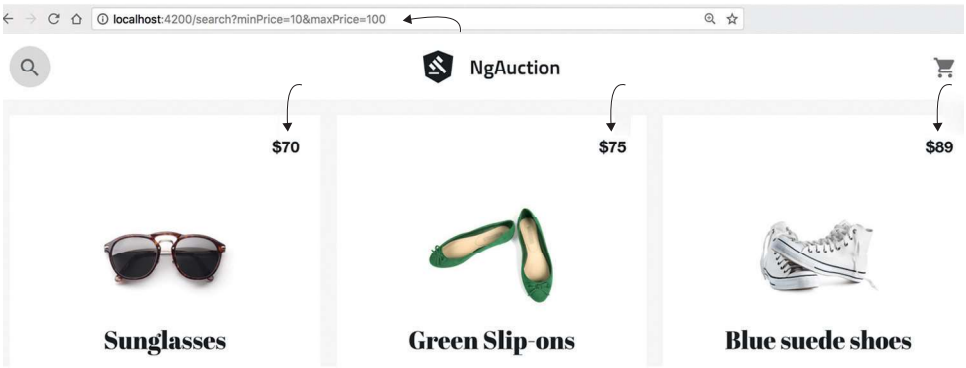


Figure 14.16 The search-result view

If the search operation initiated by the test script returns products, you extract the price of the first product, so later on (in the spec) you can assert that the product price meets the search criteria. Because the search may return a collection of products, you'll access them using the alias `$$` for the `element.all` API.

Each of the products has the `tile__price-tag` style, as shown in figure 14.17, taken from the Element tab in the Chrome Dev Tools panel while the products grid was shown. You'll use the `tile__price-tag` style to locate products.

```
▼<nga-search _nghost-c13 class="ng-star-inserted">
  ▼<div _ngcontent-c13 class="grid-list-container">
    ▼<nga-product-grid _ngcontent-c13 _nghost-c10 ng-reflect-products="[object Object],[object Object">
      ▼<mat-grid-list _ngcontent-c10 class="mat-grid-list" guttersize="16" ng-reflect-cols="3" ng-reflect-gutter-size="16"
      style="padding-bottom: calc(((4 * ((33.3333% - 10.6667px) * 1)) + 0px) + 48px);">
        ▼<div>
          <!--bindings={
            "ng-reflect-ng-for-of": "[object Object],[object Object"
          }-->
          ▼<mat-grid-tile _ngcontent-c10 class="tile mat-grid-tile ng-star-inserted" style="left: 0px; width: calc(((33.3333% -
          10.6667px) * 1) + 0px); margin-top: 0px; padding-top: calc(((33.3333% - 10.6667px) * 1) + 0px);">
            ▼<figure class="mat-figure">
              ▼<a _ngcontent-c10 class="tile__content" fxlayout="column" fxlayoutalign="center center" ng-reflect-router-link="/
              products,2" ng-reflect-layout="column" ng-reflect-align="center center" href="/products/2" style="flex-direction:
              column; box-sizing: border-box; display: flex; max-width: 100%; place-content: center; align-items: center;">
                <span _ngcontent-c10 class="tile__price-tag" ngclass.xs="tile__price-tag--xs" ng-reflect-klazz="tile__price-tag"
                ng-reflect-ng-class-xs="tile__price-tag--xs">
                  $70
                </span>
```

Figure 14.17 CSS selector for the price

When the product price is extracted, you need to convert it to a number. In ngAuction, the product price is rendered as a string with the dollar sign, such as "$70" in figure 14.17. But you need its numeric representation so the spec can assert that the price falls within the specified range. The `getFirstProductPrice()` method includes the code that removes the dollar sign from the string and converts it to an integer value, as you can see in the next listing.

Listing 14.30 Getting the price of the first product

```
getFirstProductPrice() {
  return $$('span[class="tile__price-tag"]')
      .first().getText()
      .then((value) => {
          return parseInt(value.replace('$', ''), 10);
      });
}
```

Uses element.all for finding products

Gets the text of the first product

Protractor's API returns promises, so applies then()

Converts the product price to a number and returns it

The complete code of your page object is shown in the following listing.

Listing 14.31 search.po.ts

```
import {protractor, browser, by, element, $, $$} from 'protractor';

export class SearchPage {

  performSearch(minimalPrice: number, maximumPrice: number) {
     const searchOnToolbar = element(by.id('search'));
     searchOnToolbar.click();

     const minPrice = $('input[formControlName="minPrice"]');
     const maxPrice = $('input[formControlName="maxPrice"]');
     minPrice.sendKeys(minimalPrice);
     maxPrice.sendKeys(maximumPrice);

     const searchOnForm = element(by.buttonText('SEARCH'));
     searchOnForm.click();
     const EC = protractor.ExpectedConditions;
     const urlChanged = EC.urlContains('/search');
      browser.wait(urlChanged, 5000, 'The URL should contain "/search"');
  }

  navigateToLandingPage() {
     return browser.get('/');
  }

  getFirstProductPrice() {
    return $$('span[class="tile__price-tag"]')
        .first().getText()
        .then((value) => {return parseInt(value.replace('$', ''), 10);});
  }
}
```

Clicks the Search icon in the toolbar

Fills out min and max prices on the search form

Clicks the Search button on the form

Declares an expected condition

Waits for the expected condition for up to 5 seconds

Declares a method for navigating to the landing page

Locates all price elements

Singles out the first product price

Converts the price into a number

Now let's review the code of the test suite located in the search.e2e-spec.ts file.

The test suite for the search workflow contains one spec, which uses the page object and adds assertions to each step of the workflow. The spec starts by navigating to the landing page of ngAuction and then asserts that the URL of the page contains the segment /categories/all.

Then the spec performs the test by invoking the `performSearch()` method on the page object, passing `10` and `100` as a price range for the search. After this method completes, it performs three assertions to check that the URL of the resulting page contains the segment /search?minPrice=10&maxPrice=100 and the price of the first product is greater than $10 and less than $100. The code of this test suite is shown in the following listing.

Listing 14.32 search.e2e-spec.ts

```
import {SearchPage} from './search.po';
import {browser} from 'protractor';

describe('ngAuction search', () => {
  let searchPage: SearchPage;

  beforeEach(() => {
    searchPage = new SearchPage();
  });

  it('should perform the search for products that cost from $10 to $100',
      () => {
    searchPage.navigateToLandingPage();
    let url =  browser.getCurrentUrl();
    expect(url).toContain('/categories/all');

    searchPage.performSearch(10, 100);
    url =  browser.getCurrentUrl();
    expect(url).toContain('/search?minPrice=10&maxPrice=100');

    const firstProductPrice = searchPage.getFirstProductPrice();
    expect(firstProductPrice).toBeGreaterThan(10);
    expect(firstProductPrice).toBeLessThan(100);
  });
});
```

Instantiates the page object

Asserts the URL of the landing page

Searches for products

Asserts the URL of the page with the search results

Asserts that the price is greater than 10

Asserts that the price is less than 100

Gets the price of the first product

In the Terminal window, switch to the client directory, run `npm install`, and run the test with the `ng e2e` command. The test will successfully complete, and you'll see the message shown in figure 14.18.

```
  ngAuction search
    ✓ should perform the search for products that cost from $10 to $100

Executed 1 of 1 spec SUCCESS in 2 secs.
```

Figure 14.18 The product-search test succeeded.

To make the test fail, modify the spec to test the case when no products are returned by using a price range between $1 and $5,000,000. Your ngAuction isn't created for Sotheby's, and you don't carry expensive items.

Summary

- Unit tests run quickly, but most application business logic should be tested with E2E tests.
- While you're writing tests, make them fail to see that their failure report is easy to understand.
- Running unit tests should be part of your automated build process, but E2E tests shouldn't.

15 Maintaining app state with ngrx

This chapter covers

- A brief introduction to the Redux data flow
- Maintaining your app state using the ngrx library
- Exploring another implementation of the Mediator design pattern
- Implementing state management in ngAuction with ngrx

You've made it to the last chapter, and you're almost ready to join an Angular project. The previous chapter was easy reading, but this chapter will require your full attention; the material we're about to present has many moving parts, and you'll need to have a good understanding of how they play together.

ngrx is a library that can be used for managing state in Angular apps (see https://github.com/ngrx). It's built using the principles of Redux (another popular library for managing state), but the notification layer is implemented using RxJS. Although Angular has other means for managing app state, ngrx is gaining traction in mid- and large-size apps.

Is it worth using the ngrx library for managing state in your app? It certainly has benefits, but they don't come free. The complexity of your app can increase, and the code will become more difficult to understand by any new person who joins the project. In this chapter, we cover the ngrx library so you'll be able to decide whether it's the right choice for managing the state of your app. In the hands-on section, we do a detailed code overview of yet another version of ngAuction that uses ngrx for state management.

15.1 From a convenience store to Redux architecture

Imagine that you're a proud owner of a convenience store that sells various products. Remember how you started? You rented an empty place (the store was in its *initial state*). Then you purchased shelves and ordered products. After that, multiple vendors started delivering those products. You hired employees who arranged these products on the shelves in a certain order, changing the state of the store. Then you put out the Grand Opening sign and festooned the place with lots of colorful balloons. Customers started visiting your store to buy products.

When the store is open, some products lay on the shelves, and some are in shopping carts of customers. Some customers are waiting in lines at cash registers, where there are store employees. You can say that at any given moment, your store has the *current state*.

If a customer takes an *action*, such as buying five bottles of water, the cashier scans the barcode, and this *reduces* the number of bottles in the inventory—it updates the state. If a vendor delivers new products, your clerk updates the inventory (state) accordingly.

Your web app can also maintain a store that holds the state of your app. Like a real store, at any given time your app's *store* has a current *state*. Some data collections have specific data retrieved from the server and possibly modified by a user. Some radio buttons are checked, and a user selects some products and navigates to some routes represented by a specific URL.

If a user interacts with the UI, or the server sends new data, these *actions* should ask the store object to update the state. To keep track of state changes, the current state object is never updated, but a new instance of the state object is created.

15.1.1 What's Redux?

Redux is an open source JavaScript library that offers a state container for JavaScript apps (see http://mng.bz/005X). It was created at Facebook as an implementation of the Flux architecture (see http://mng.bz/jrXy). Initially, the developers working with the React framework made Redux popular, but as it's a JavaScript library, it can be used in any JavaScript app.

Redux is based on the following three principles:

- *There's a single source of truth.* There's a single store where your app contains the state that can be represented by an object tree.

- *State is read-only.* When an action is emitted, the reducer function doesn't update but clones the current state and updates the cloned object based on the action.
- *State changes are made with pure functions.* You write the reducer function(s) that take an action and the current state object and return a new state.

In Redux, the data flow is unidirectional:

1. The app component dispatches the action on the store.
2. The reducer (a pure function) takes the current state object and then clones, updates, and returns it.
3. The app component subscribes to the store, receives the new state object, and updates the UI accordingly.

Figure 15.1 shows the unidirectional Redux data flow.

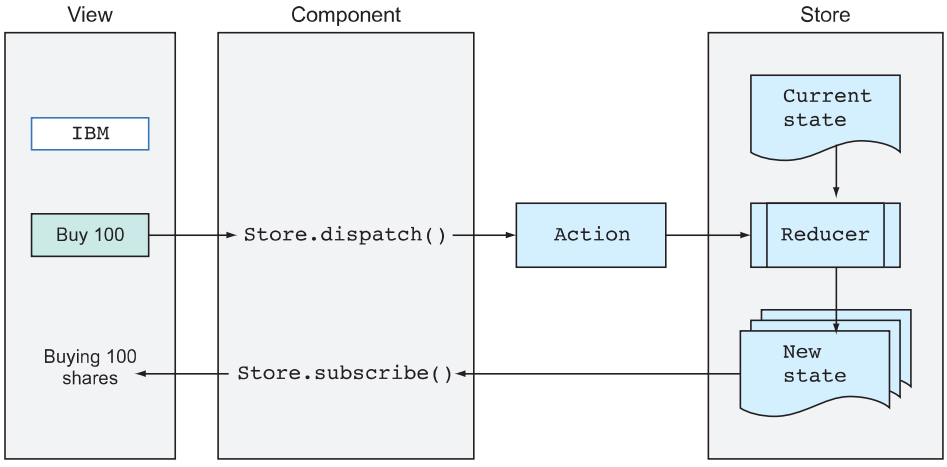


Figure 15.1 The Redux data flow

An *action* is a JavaScript object that has a `type` property describing what happens in your app, such as a user wants to buy IBM stock. Besides the `type` property, an action object can optionally have another property with a payload of data that should change the app state in some fashion. An example is shown in the following listing.

Listing 15.1 An action to buy IBM stock

```
{
  type: 'BUY_STOCK',
   stock: {symbol: 'IBM', quantity: 100}
}
```

The type of action

The action payload

This object only describes the action and provides the payload. It doesn't know how the state should be changed. Who does? The reducer.

A *reducer* is a *pure function* that specifies how the state should be changed. The reducer never changes the current state, but creates a new (and updated) version of it. The state object is immutable. The reducer creates a copy of the state object and returns a new reference to it. From an Angular perspective, it's a binding change event, and all interested parties will immediately know that the state has changed without requiring expensive value checking in the entire state tree.

> **NOTE** Your state object can contain dozens of properties and nested objects. Cloning the state object creates a shallow copy without copying each unmodified state property in memory, so memory consumption is minimal and it doesn't take much time. You can read about the rationale for creating shallow state copies at http://mng.bz/3271.

A reducer function has the signature shown in the following listing.

Listing 15.2 A reducer signature

```
function (previousState, action): State {...}
```

A reducer function returns a new state.

Should the reducer function implement app functionality like placing an order, which requires work with external services? No, because reducers are meant for updating and returning the app state—for example, the stock to buy is `"IBM"`. Implementing app logic would require interaction with the environment external to the reducer; it would cause *side effects*, and pure functions can't have side effects.

The reducer can implement minimal app logic related to state change. For example, suppose a user decides to cancel an order, which requires a reset of certain fields on the state object. The main app logic remains in your application code (for example, in services) unless a concrete implementation of the Redux-inspired library offers a special place meant for code with side effects. In this chapter, we use the ngrx library, which suggests using Angular services combined with so-called *effects* that live outside of the store and that can aggregate Angular services working as a bridge between the store and services.

15.1.2 Why storing app state in a single place is important

Recently, one of the authors of this book worked on a web project for a large car manufacturer. This was a web app that allowed a prospective buyer to configure a car by selecting from more than a thousand packages and options (such as model, interior and exterior colors, length of chassis, and so on). The app was developed over many years. The software modules were written using JavaScript, jQuery, Angular, React, and Handlebars, as well as the HTML templating engine Thymeleaf on the server.

From a user perspective, this was one workflow that consisted of several steps resulting in configuring and pricing the car based on selected options. But internally, the process was switching from one module to another, and each module needed to know what was selected in the previous step to show the available options.

In other words, each module needed to know the current state of the app. Depending on the software used in any particular module, the current user selections were stored using one of the following:

- URL parameters
- HTML `data*` attributes
- The browser's local and session storage
- Angular services
- The React store

New requirements came in, new JIRA tickets were created and assigned, and implementation would begin. Time and time again, implementing a seemingly simple new requirement would turn into a time-consuming and expensive task. Good luck explaining to the manager why showing the price in page B would take a half day even though this price was already known in page A, or that the state object used in page B didn't expect to have the price property, and if in page A the price was a part of the URL, page B expected to get the current state from local storage. Rewriting it from scratch was not an option. It would have been so much easier if app state had been implemented in a uniform way and stored in a single place!

TIP If you're starting to develop a new project, pay special attention on how app state is implemented, and it will help you greatly in the long run.

15.2 Introducing ngrx

ngrx is a library inspired by Redux. You can think of it as an implementation of the Redux pattern for managing app state in Angular apps. Similarly to Redux, it implements a unidirectional data flow and has a store, actions, and reducers. It also uses RxJS's ability to send notifications and subscribe to them.

Large enterprise apps often implement messaging architecture on the server side, where one piece of software sends a message to another via some messaging server or a message bus. You can think of ngrx as a client-side messaging system. The user clicks a button, and the app sends a message (for example, dispatches an action). The app state changed because of this button click, and the ngrx `Store` sends a message to the subscriber(s), emitting the next value into an observable stream.

In section 15.1, we described three Redux principles:

- A single source of truth.
- State is read-only.
- State changes are made with pure functions.

In ngrx, app state is accessed with the `Store` service, which is an observable of state and an observer of actions. The declaration of the class `Store` in the store.d.ts file looks like this:

```
class Store<T> extends Observable<T> implements Observer<Action>
```

Besides declaring a new principle, the ngrx architecture includes *effects*, which are meant for the code that communicates with other parts of the app, such as making HTTP requests. With ngrx selectors, you can subscribe to changes in a particular branch of the state object. There's also support for routing and collections of entities, which can be useful in CRUD operations.

We'll start our ngrx introduction with its main players: a store, actions, and reducers.

15.2.1 Getting familiar with a store, actions, and reducers

Let's see how to use ngrx in a simple app that has two buttons that can either increment or decrement the value of the counter. The first version of this app doesn't manage state and looks like the following listing.

Listing 15.3 The counter app without ngrx

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

@Component({
  selector: 'app-root',
  template: `
     <button (click)="increment()">Increment</button>
     <button (click)="decrement()">Decrement</button>
     <p>The counter: {{counter}}</p>          <-- Shows the counter value
  `
})
export class AppComponent {
  counter = 0;

  increment() {
    this.counter++;      <-- Increments the counter
   }

  decrement() {
    this.counter--;      <-- Decrements the counter
   }
}
```

You want to change this app so that the ngrx store manages the state of the `counter` variable, but first you need to install the ngrx store in your project:

```
npm i @ngrx/store
```

The `Store` serves as a container of the state, and dispatching actions is the only way to update the state. The plan is to instantiate the `Store` object and remove the app logic (incrementing and decrementing the counter) from the component. Your `decrement()` and `increment()` methods will be dispatching actions on the `Store` instead.

Actions are handled by the ngrx reducer, which will update the state of the counter. The type of your `counter` variable will change from `number` to `Observable`, and to get and render its emitted values in the UI, you'll subscribe to the `Store`.

The only required property in the `Action` object is `type`, and for your app, you'll declare the action types as follows:

```
const INCREMENT = 'INCREMENT';
const DECREMENT = 'DECREMENT';
```

The next step is to create a reducer function for each piece of data you want to keep in the store. In your case, it's just the value of the counter, so you'll create a reducer with the `switch` statement for updating state based on the received action type, as shown in the following listing. Remember, the reducer function takes two arguments: state and action.

Listing 15.4 reducer.ts

```
import { Action } from '@ngrx/store';

export const INCREMENT = 'INCREMENT';
export const DECREMENT = 'DECREMENT';

export function counterReducer(state = 0, action: Action) {
     switch (action.type) {
         case INCREMENT:
            return state + 1;

         case DECREMENT:
            return state - 1;

         default:
            return state;
     }
}
```

The initial value of the counter (state) is zero.

Checks the action type

Updates state by incrementing the counter

Updates state by decrementing the counter

Returns the existing state if an unknown action is provided

It's important to note that the reducer function doesn't modify the provided state, but returns a new value. The state remains immutable.

Now you need to inform the root module that you're going to use the `counterReducer()` function as a reducer for your store, as shown in the following listing.

Listing 15.5 app.module.ts

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';
import {AppComponent} from './app.component';
import {counterReducer} from "./reducer";
import {StoreModule} from "@ngrx/store";

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    StoreModule.forRoot({counterState: counterReducer})
   ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Lets the store know about the reducer for the app

In this code, you configure the app-level store to provide the object that specifies `counterReducer` as the name of the reducer function, and `counterState` as the property where this reducer should keep the state.

Finally, you need to change the code of your component to dispatch either the action of type `INCREMENT` or `DECREMENT`, depending on which button a user clicks. You'll also inject the `Store` into your component and subscribe to its observable that will emit a value each time the counter changes, as shown in the following listing.

Listing 15.6 app.component.ts

```
import {Component} from '@angular/core';
import {Observable} from "rxjs";
import {select, Store} from "@ngrx/store";
import {INCREMENT, DECREMENT} from "./reducer";

@Component({
  selector: 'app-root',
  template: `
     <button (click)="increment()">Increment</button>
     <button (click)="decrement()">Decrement</button>
     <p>The counter: {{counter$ | async}}</p>
   `
})
export class AppComponent {
  counter$: Observable<number>;

  constructor(private store: Store<any>) {
    this.counter$ = store.select('counterState');
   }

  increment() {
    this.store.dispatch({type: INCREMENT});
   }

  decrement() {
    this.store.dispatch({type: DECREMENT});
   }
}
```

Subscribes to the observable with the async pipe

Declares the reference variable for the store observable

select() emits changes in the counterState.

Dispatches the INCREMENT action

Dispatches the DECREMENT action

Note that an action is an object (for example, `{type: INCREMENT}`), and in this app, action objects have no payload. You can also think of an action as a message or a command. In the next section, you'll be defining each action as a class with two properties: type and payload.

In this component, you use the `select` operator (defined in the ngrx `Store`), which allows you to observe the state object. The name of its argument must match the name of the state object property used in the `StoreModule.forRoot()` function .

NOTE The `counterReducer` was assigned to the store by invoking the method `StoreModule.forRoot({counterState: counterReducer})` in the module. The `AppComponent` communicated with the `counterReducer` either by dispatching an action on the store or by using the `select` operator on the store.

The app with the ngrx store will have the same behavior as the original one and will increment and decrement the counter depending on the user's action.

Now let's check whether your store is really a single source of truth. You'll add a child component in the next listing that will display the current value of the counter received from the store, and 10 seconds after app launch, the child will dispatch the INCREMENT action.

Listing 15.7 child.component.ts

```
import {Component} from '@angular/core';
import {select, Store} from "@ngrx/store";
import {Observable} from "rxjs";
import {INCREMENT} from "../reducer";

@Component({
  selector: 'app-child',
  template: `
    <h3> Child component </h3>
    <p>
      The counter in child is {{childCounter$ | async}}
    </p>
  `,
  styles: []
})
export class ChildComponent {

    childCounter$: Observable<number>;

    constructor(private store: Store<any>) {
         this.childCounter$ = store.pipe(select('counterState'));

        setTimeout(() => this.store.dispatch({type: INCREMENT}),
             10000);
    }
}
```

Injects the store

Subscribes to the store

In 10 seconds, dispatches the INCREMENT action

The only thing left is adding the <app-child> tag to the template of AppComponent. Figure 15.2 shows the app after the user clicks the Increment button three times. Both parent and child components show the same value of the counter taken from the store (single source of truth). Ten seconds after the app starts, the ChildComponent dispatches the INCREMENT action, and both components will show the incremented counter.

To see this app in action, open the project counter, run npm install, and then run ng serve -o.

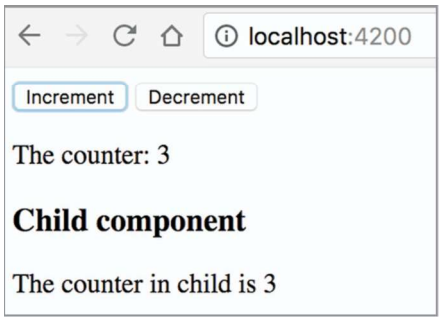


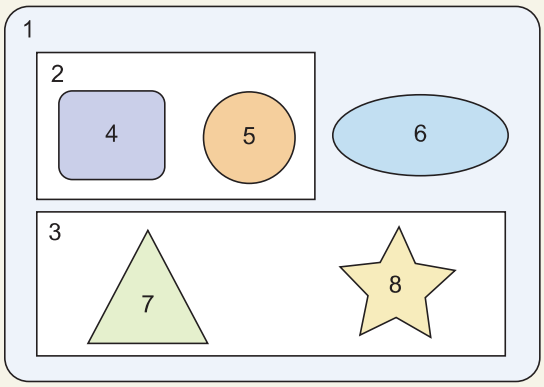
Figure 15.2 Running the counter app

NOTE The ngrx library includes the example app (see http://mng.bz/7F9x), which allows you to maintain a book collection using Google Books API. The ngAuction app that comes with this chapter can also serve as an ngrx demo, although neither of these apps uses every API offered by ngrx.

The counter app is a pretty basic example, with a single reducer function. In practice, a store may have several reducers where each of them would be responsible for a portion of the state object. In the hands-on section, the new version of ngAuction will have several reducers.

Eliminating the need for event bubbling

Here's a diagram you saw in chapter 8 in section 8.3.1.



A view consists of components

Say component 7 can emit some data that's needed in component 6. If you use the common parents for intercomponent communication, you need to emit an event via the `@Output` property of component 7; parent component 3 would subscribe to this event and reemit it through its `@Output` property; and component 1 would subscribe to this event and, via binding, pass the payload to component 6.

Using the ngrx store eliminates the need to program this series of unfortunate events. Component 7 emits an action of the store, and component 6 uses a selector to receive it. The same, simple model of intercomponent communication works regardless of how many levels of component nesting exist in any particular view. The only thing that both components 7 and 6 need is the reference to the store object.

This figure doesn't give any details about what these eight components do, but you can assume that 1, 2, and 3 are *container components*, which include other components and implement app logic for interacting with their children, parents, and services. The rest are *presentational components* that can get data in, send data out, and present data on the UI. Some authors suggest that only container components should manage state and communicate with the store. We don't agree with this, because state is not only about storing and sharing data—it's also about storing the state of the UI, which is a part of any type of component.

In the counter example, the store managed app state represented by a number, but did you notice the store played yet another role? In section 8.3.2 in chapter 8, we showed you how an injectable service can play the role of mediator. In the counter app, the main goal of the ngrx store is to manage app state, but it also plays another role: serving as a mediator between parent and child components.

In chapter 8, the mediator was a service with an RxJS `BehaviorSubject`, and you used components for sending and receiving data. With ngrx, you don't need to manually create `BehaviorSubject`, because the `Store` object can be used for emitting values as well as subscribing to them.

To notify `BehaviorSubject` about the new value, you use `next()`, and to notify the store about the new state, you use `dispatch()`. To get the new state, subscribe to the observable in both cases. Figure 15.3 compares the code of `EbayComponent` from listing 8.13 in chapter 8 (on the left) with `ChildComponent` that uses ngrx (on the right). They look similar, don't they?

```
@Component({
  selector: 'product',
  template: `<div>
              <h2 >eBay component</h2>
             Search criteria: {{searchFor$ | async}}
             </div>`
})
export class EbayComponent {

  searchFor$: Observable<string>;

  constructor(private state: StateService) {

      this.searchFor$ = state.getState();
  }
}
```

```
@Component({
  selector: 'app-child',
  template: `
    <h3> Child component </h3>
    <p>The counter in child is {{childCounter$ | async}}
    </p>`
})
export class ChildComponent {

  childCounter$: Observable<number>;

  constructor(private store: Store<any>) {

     this.childCounter$ = store.pipe(select('counterState'));
  }
}
```

Figure 15.3 `EbayComponent` compared to `ChildComponent`

We can say that the `StateService` (left) and the `Store` (right) each serve as a single source of truth. But large non-ngrx apps with multiple injectable services that store different slices of state would have multiple sources of truth. In ngrx apps, the `Store` service always remains a single source of truth, which may have multiple slices of state.

Now take another look at the reducer in listing 15.4, which was a pure function that didn't need to use any external resource to update the state. What if the value for a counter was provided by a server? The reducer can use external resources because it would make the reducer *impure*, wouldn't it? This is where ngrx effects come in, and we'll discuss them next.

15.2.2 Getting familiar with effects and selectors

Reducers are *pure functions* that perform simple operations: take the state and action and create a new state. But you need to implement business logic somewhere, such as reaching out to services, making requests to servers, and so on. You need to implement *functions with side effects*, which is done in effect classes.

Effects are injectable classes that live outside of the store and are used for implementing functionality that has side effects, without breaking unidirectional data flow. ngrx effects come in a separate package, and you need to run the following command to add them to your project:

```
npm i @ngrx/effects
```

If a component dispatches an action that requires communication with external resources, the action can be picked up by the `Effects` object, which will handle this action and dispatch another one on the reducer. For example, an effect can receive a `LOAD_PRODUCTS` action from the store, invoke `loadProducts()`, and, when the data is loaded, dispatch either of the `LOAD_PRODUCTS_SUCCESS` or `LOAD_PRODUCTS_FAILURE` actions. The reducer will pick it up and update state accordingly. Figure 15.4 shows the ngrx flow that uses effects.

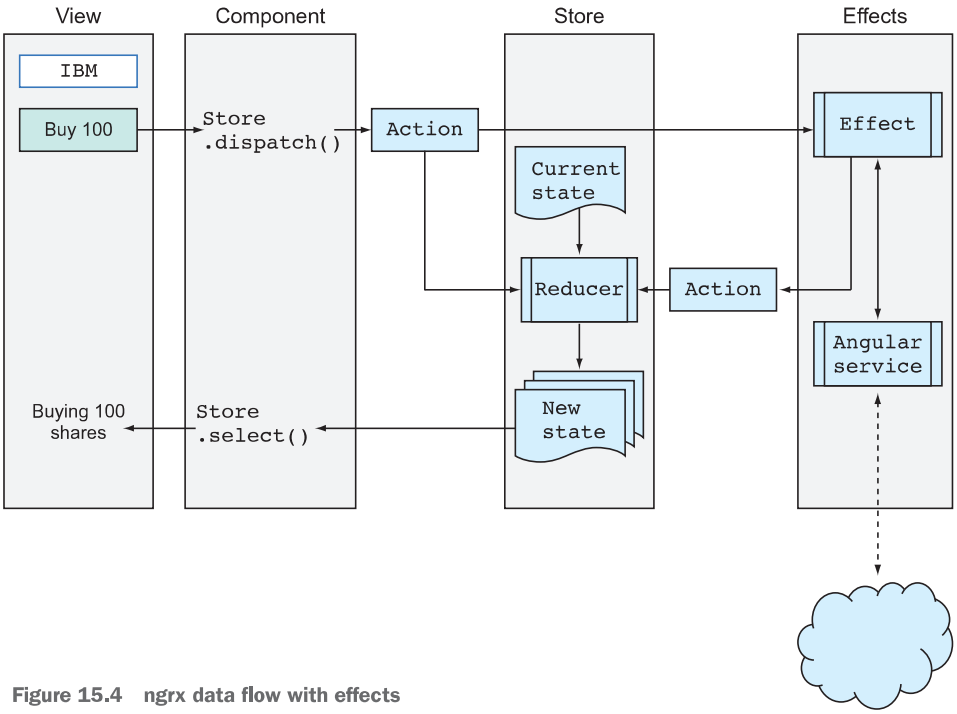


Figure 15.4 ngrx data flow with effects

To understand this diagram, imagine that a user clicks the Buy 100 button. The component would dispatch an action on the store, which can be handled by a reducer, an effect, or both. An effect can access external services and dispatch another action. In any case, a reducer is ultimately responsible for creating a new state, and a component can get it using a selector and update the UI accordingly (such as by rendering the message "Buying 100 shares").

NOTE We'd like to stress that even though actions can be handled in both a reducer and an effect, only a reducer can change the state of an app.

If you compare the Redux and ngrx data flows shown in figures 15.4 and 15.1 respectively, you'll notice that the effects live outside the store. They can communicate with other Angular services, which in turn can communicate with external servers, if need be. Another difference in figure 15.1 is that the view would use `subscribe()` to receive the latest state; 15.4 shows the `select()` method that can use a selector function to retrieve either the entire state object or its part.

In both Redux and ngrx, a component dispatches actions on the store. Redux actions are handled only in reducers, but in ngrx, some actions are handled in reducers, some in effects, and some in both. For example, if a component dispatches `LOAD_PRODUCTS`, a reducer can pick it up to set the state property `loading` to true, which will result in displaying a progress indicator. An effect can receive the same `LOAD_PRODUCTS` action and make an HTTP request for products.

You know that to dispatch an action that should be handled by the reducer, a component invokes `Store.dispatch()`, but how can an effect dispatch an action? An effect returns an observable that wraps some payload. In your effects class, you'll declare one or more class variables annotated with the `@Effect` decorator. Each effect will apply the `ofType` operator to ensure that it reacts to only the specified action type, as shown in the following listing.

Listing 15.8 A fragment of a class with effects

```
@Injectable()
export class MyEffects {
...
@Effect()
loadProducts$ = this.actions$
.pipe(ofType(LOAD_PRODUCTS),
.switchMap(this.productService.getProducts()))
...
}
```

In this example, the `@Effect` decorator marks the observable property `loadProducts$` as a handler for the actions of type `LOAD_PRODUCTS` and invokes `getProducts()`, which returns an `Observable`. Then, based on the emitted value, the effect will dispatch another action (for example, success or failure). You'll see how to do this in the next section. In general, you can think of an effect as middleware between the original action and the reducer, as shown in figure 15.5.

In your app module class, you need to add to the `@NgModule` decorator `EffectsModule.forRoot()` for the root module, or `EffectsModule.forFeature()` for a feature module.

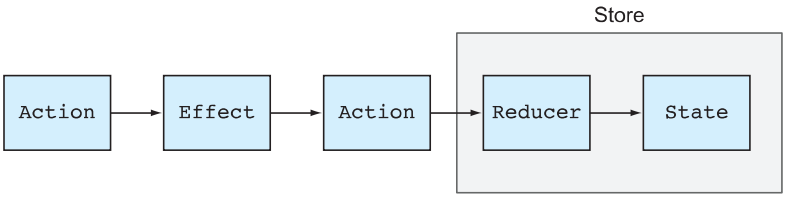


Figure 15.5 Effects in the data flow

We don't want to overwhelm you with the theory behind ngrx, so let's continue developing an app that uses an ngrx store, actions with payloads, reducers, effects, and selectors.

15.2.3 Refactoring the mediator app with ngrx

In this section, you'll refactor the app created in section 8.3.2 in chapter 8. That app had a search field and two links, eBay and Amazon. You'll refactor it by replacing the `SearchService` injectable that was maintaining app state with an ngrx store. To illustrate communication between effects and services, you'll add `ProductService`, which will generate the search results: the products containing the entered search criterion in their names.

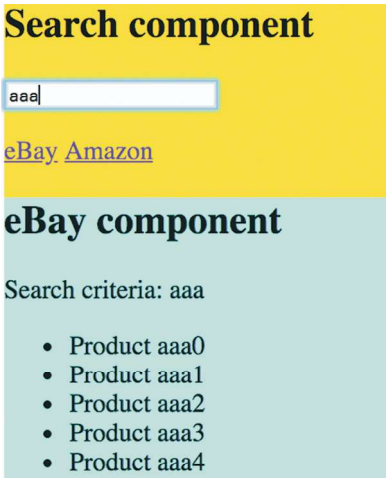


Figure 15.6 The results of an `aaa` search on eBay

The new version of the app is located in the mediator folder. It will use the following ngrx building blocks:

- A store for storing and retrieving app state, search query, and results
- A reducer for handing actions of type `SEARCH` and `SEARCH_SUCCESS`
- Effects for handling actions of type `SEARCH` and `SEARCH_SUCCESS`
- Selectors for retrieving the entire state object, search query, or search results

Figure 15.6 shows the mediator app after a user enters `aaa` in the search field of the `SearchComponent`.

THE STORE STATE

The state object of this app will contain two properties: search query (for example, `aaa`) and search results (for example, five products). You'll declare the type in the following listing to represent the state of your app.

Listing 15.9 The state of the mediator app

```
export interface State {
  searchQuery: string;
   searchResults: string[];
 }
```

Payload of the SEARCH action dispatched by SearchComponent

Payload of SEARCH_SUCCESS dispatched by the effect after invoking ProductService.getProducts()

ACTIONS

In the counter app, actions don't contain payloads; they increment or decrement the counter. This time it's different. The `SEARCH` action can have a payload (such as `aaa`), and `SEARCH_SUCCESS` can have a payload as well (such as five products). That's why declaring constants representing the action type isn't enough, and you'll wrap each action into a class with a constructor that has a payload as an argument. The actions will be declared in the file actions.ts, shown in the following listing.

Listing 15.10 actions.ts

```
import {Action} from '@ngrx/store';

export const SEARCH = '[Product] search';
 export const SEARCH_SUCCESS = '[Product] search success';

export class SearchAction implements Action {
   readonly type = SEARCH;

  constructor(public payload: {searchQuery: string}) {}
}

export class SearchSuccessAction implements Action {
   readonly type = SEARCH_SUCCESS;

  constructor(public payload: {searchResults: string[]}) {}
}

export type SearchActions = SearchAction | SearchSuccessAction;
```

Declares action types

The class representing the search action with a payload

The class representing the search-success action with a payload

Declares the union SearchAction type

Note the text `[Product]` in the action definitions. In real-world apps, you may have more than one `SEARCH` action—one for products, one for orders, and so on. By prepending the action description with `[Product]`, you create a namespace to make the code more readable. Having namespaced actions helps in understanding which actions were dispatched at any given moment.

The last line of actions.ts uses the TypeScript union operator described in section B.11 in appendix B. Here, you define the `SearchActions` type that will be used in the reducer's signature, so the TypeScript compiler knows which actions are allowed in the reducer's `switch` statement.

THE SEARCHCOMPONENT AS ACTION CREATOR

The actions are declared, but someone has to create and dispatch them. In your app, the `SearchComponent` shown in the following listing will be creating and dispatching the action of type `SEARCH` after the user enters the search criterion.

Listing 15.11 search.component.ts

```
@Component({
  selector: 'app-search',
  template: `
    <h2>Search component</h2>
    <input type="text" placeholder="Enter product"
                       [formControl]="searchInput">`,
  styles: ['.main {background: yellow}']
})
export class SearchComponent {

  searchInput: FormControl;

  constructor(private store: Store<any>) {
    this.searchInput = new FormControl('');

    this.searchInput.valueChanges
      .pipe(debounceTime(300),
            tap(value => console.log(`The user entered ${value}`)))
      .subscribe(searchValue => {
          this.store.dispatch(new SearchAction({ searchQuery: searchValue }))
    });
  }
}
```

Subscribes to the form control's observable

Instantiates and dispatches an action of type SEARCH with the payload

The dispatched action will be picked up by the reducer, which will update the `searchQuery` property on the state object.

> **NOTE** We'll talk about another action creator, the `SearchEffects` class, later in this section.

THE REDUCER

In the reducer shown in listing 15.12, you declare the interface describing the structure of your app state and create an object that represents an initial state. The `reducer()` function will take the initial or current immutable state and, using a `switch` statement, will create and return a new state based on the action type.

Listing 15.12 reducers.ts

```
import {SearchActions, SEARCH, SEARCH_SUCCESS} from './actions';

export interface State {
   searchQuery: string;
  searchResults: string[];
}

const initialState: State = {
```

Declares the structure of the state object

Creates an object representing the initial state

```
    searchQuery: '',
  searchResults: []
};

export function reducer(state = initialState, action: SearchActions): State {
  switch (action.type) {

    case SEARCH: {
      return {
        ...state,
        searchQuery: action.payload.searchQuery,
        searchResults: []
      }
    }

    case SEARCH_SUCCESS: {
      return {
        ...state,
        searchResults: action.payload.searchResults
      }
    }

    default: {
      return state;
    }
  }
}
```

This action is dispatched by the component.

Copies the existing state values into the new state object

Updates two state properties with the new values

This action will be dispatched by the effect.

Copies the existing state values into the new state object

Updates one state property with the new value

Returns the current state if an unexpected action was dispatched

TIP If, in a `case` clause, you use the action type that wasn't declared in the union type `SearchActions` (for example, `SEARCH22`), the TypeScript compiler returns an error.

The more precise name for TypeScript union types is *discriminated unions*. If all the types in a union have a common type property, the TypeScript compiler can discriminate types by this property. It knows which particular type from the union was referred to within the `case` statement and suggests the correct type for the payload property.

For cloning the state object and updating some of its properties, you use the spread operator described in section A.7 in appendix A. Note that state properties will be updated with the value of the action payload.

EFFECTS

In this app, you'll have one effect that will use a `ProductService` injectable to obtain products. To simplify the explanation, you don't load products from an external server or file. Your `ProductService`, shown in the following listing, will generate and return an observable of five products. It uses the RxJS `delay` operator to emulate a one-second delay as if the products are coming from a remote computer.

Listing 15.13 product.service.ts

```
@Injectable()
export class ProductService {

  static counter = 0;
```

The counter concatenated to the search query is a product name.

```
  getProducts(searchQuery: string): Observable<string[]> {

    const productGenerator = () =>
      `Product ${searchQuery}${ProductService.counter++}`;
    const products = Array.from({length: 5}, productGenerator);

    return Observable.of(products).pipe(delay(1000));
  }
}
```

A function to generate a product name

Returns the observable of products after a one-second delay

Creates a five-element array using productGenerator()

Your `SearchEffects` class will declare one effect, `loadProducts$`, that will dispatch the `SEARCH_RESULTS` effect having an array of products as its payload. You want to ensure that this effect will obtain products only if the store dispatched the `SEARCH` effect, so you use the ngrx operator `ofType(SEARCH)`.

This effect extracts the payload of the action of type `SEARCH` (the search query) emitted by the `actions$` observable, and, using `switchMap`, will pass it over to the inner observable (the `getProducts()` method). Finally, the effect will dispatch the action of type `SEARCH_RESULTS` with the payload, all of which you can see in the following listing.

Listing 15.14 effects.ts

```
@Injectable()
export class SearchEffects {

  @Effect()
  loadProducts$ = this.actions$
     .ofType(SEARCH)
     .pipe(
      map((action: SearchAction) => action.payload),
       switchMap(({searchQuery})
          => this.productService.getProducts(searchQuery)),
       map(searchResults => new SearchSuccessAction({searchResults}))
      );

  constructor(private actions$: Actions,
              private productService: ProductService) {}
}
```

Initializes the loadProducts$ effect with the stream/ observable

Executes a search only if the store dispatched the SEARCH action

Extracts the payload from the action of type SEARCH

Dispatches the action of type SEARCH_SUCCESS with its payload

Obtains the product based on the specified search query

Injects the ProductService

Injects the ngrx Actions observable

In this example, you assume that `getProducts()` will always emit products, but you could add the `catchError()` function to the observer, where you'd emit the action that reports an error. You'll see the use of `catchError()` in listing 15.31.

TIP Although it's okay to abandon unwanted results with `switchMap` while reading data, if you write an effect that performs the add, update, or delete operations, use `concatMap` instead. This will prevent possible race conditions when one request is in the middle of updating a record and another one comes in. With `concatMap`, all requests will arrive at the service one after another.

In some cases, you may want to create an effect that handles an action but doesn't need to dispatch another one. For example, you may want to create an effect that merely logs the action. In such cases, you need to pass a `{dispatch: false}` object to the `@Effect` decorator:

```
@Effect({ dispatch: false })
logAction$ = this.actions$
    .pipe(
      tap( action => console.log(action))
    );
```

SELECTORS

In real-world apps, the state object can be represented by a tree of nested properties, and you may want to obtain specific slices of the store state rather than obtain the entire state object and manually traverse its content. Let's see how app components can get the value of a specific state property by using selectors.

First, get the selector of the top-level feature state using the `createFeatureSelector()` method . Then, use this selector as a starting point for other more specific selectors using the `createSelector()` method , which returns a callback function for selecting a slice of state. The selectors of your app are declared in the file selectors.ts.

Listing 15.15 selectors.ts

```
import {createFeatureSelector, createSelector} from '@ngrx/store';
import {State} from './reducers';

export const getState = createFeatureSelector<State>('myReducer');
export const getSearchQuery = createSelector(getState,
                                  state => state.searchQuery);
export const getSearchResults = createSelector(getState,
                                  state => state.searchResults);
```

Creates a top-level selector of the top-level state

Creates a selector for the state property searchResults

Creates a selector for the state property searchQuery

The argument of the `createFeatureSelector()` method is the name of the reducer specified in the module. In the `@NgModule` decorator, you'll have the following line:

```
StoreModule.forRoot({myReducer: reducer})
```

That's why, to obtain the reference to this reducer, you write `createFeatureSelector ('myReducer');`.

Let's recap what you've accomplished so far:

1. You declared classes to represent the actions of types `SEARCH` and `SEARCH_RESULTS`.
2. The `SearchComponent` can dispatch the action of type `SEARCH`.
3. The reducer can handle both action types.
4. You declared the effect that can obtain products and dispatch the action of type `SEARCH_RESULTS`.
5. You declared selectors to obtain slices of the app state.

To close the loop, you'll use the selectors in eBay and Amazon components to render the search criterion and the retrieved products. The following listing shows only the code of the `EbayComponent` (the code of the `AmazonComponent` looks identical).

Listing 15.16 ebay.component.ts

```
@Component({
  selector: 'app-ebay',
  template: `
    <div class="ebay">
      <h2>eBay component</h2>
      Search criteria: {{searchFor$ | async}}

      <ul>
        <li *ngFor="let p of searchResults$ | async ">{{ p }}</li>
       </ul>
    </div>`,
  styles: ['.ebay {background: cyan}']
})
export class EbayComponent {

  searchFor$ = this.store.select(getSearchQuery);

  searchResults$ = this.store.select(getSearchResults);

  constructor(private store: Store<State>) {}
 }
```

Subscribes to the observable that emits the search criteria and renders it

Subscribes to the observable that emits products and renders them

Invokes the getSearchQuery() selector on the store

Invokes the getSearchResults() selector on the store

Injects the store

The code of `EBayComponent` is concise and doesn't contain any app logic. With ngrx, you need to write more code, but each method in your Angular component becomes either a command that sends an action or a selector that retrieves the data, and each command changes the state of your app.

There's one more step to complete the app-ngrx communication. You need to register the store and effects in the app module. Your module, shown in the next listing, also includes route configuration, so a user can navigate between the eBay and Amazon components.

Listing 15.17 app.module.ts

```
@NgModule({
  imports: [BrowserModule, CommonModule, ReactiveFormsModule,
    RouterModule.forRoot([
      {path: '',        component: EbayComponent},
       {path: 'amazon', component: AmazonComponent}]),

    StoreModule.forRoot({myReducer: reducer}),
     EffectsModule.forRoot([SearchEffects]),
         StoreDevtoolsModule.instrument({
                   logOnly: environment.production}),
   ],
  declarations: [AppComponent, EbayComponent, AmazonComponent, SearchComponent],
  providers: [
    ProductService,
    {provide: LocationStrategy, useClass: HashLocationStrategy}
  ],
  bootstrap:[AppComponent]
})
export class AppModule {}
```

Configures the routes

Registers the effects

Enables the use of Redux DevTools

Registers the store and links it to the reducer

In the next section, we'll show you how to monitor state with the Chrome extension Redux DevTools and what the `instrument()` method is for.

The app component in the following listing remains the same as in the mediator example from chapter 8. Listing 8.10 contains annotations, so we won't describe it here.

Listing 15.18 app.component.ts

```
@Component({
  selector: 'app-root',
  template: ` <div class="main">
              <app-search></app-search>
              <p>
              <a [routerLink]="['/']">eBay</a>
              <a [routerLink]="['/amazon']">Amazon</a>
              <router-outlet></router-outlet>
              </div>`,
  styles: ['.main {background: yellow}']
})
export class AppComponent {}
```

To see this app in action, run `npm install` in the project mediator, and then run `ng serve -o`.

What else ngrx has to offer

Your mediator app utilizes the packages `@ngrx/store` and `@ngrx/effects`, which can address most of your state-management needs. In the hands-on section, you'll also use `@ngrx/router-store`, which offers bindings for connecting and monitoring Angular `Router`. There are other packages as well:

- `@ngrx/entity` is an entity state adapter for managing record collections.
- `@ngrx/schematics` is a scaffolding library that provides blueprints for generating ngrx-related code.

Consider exploring these packages on your own. The API of all ngrx packages is described at http://mng.bz/362y.

Now let's see how to monitor app state with Redux DevTools.

15.2.4 Monitoring state with ngrx store DevTools

Because you delegate state-management operations to ngrx, you need a tool to monitor state changes during runtime. The browser extension Redux DevTools along with the `@ngrx/store-devtools` package are used for the instrumentation of the app state. First, install `@ngrx/store-devtools`:

```
npm install @ngrx/store-devtools
```

Second, add the Chrome extension Redux DevTools (there is such an add-on for Firefox as well).

Third, add the instrumentation code to the app module. For example, for instrumentation with the default configuration, you can add the following line to the imports section of the `@NgModule` decorator:

```
StoreDevtoolsModule.instrument()
```

`StoreDevtoolsModule` must be added after `StoreModule`. If you want to add instrumentation minimizing its overhead in the production environment, you can use the `environment` variable as follows:

```
StoreDevtoolsModule.instrument({
  logOnly: environment.production
})
```

In production, set the `logOnly` flag to `true`, which doesn't include tools like dispatching and reordering actions, persisting state and actions history between page reloads that introduces noticeable performance overhead. You can find the complete list of features that `logOnly: true` turns off at http://mng.bz/cOwC.

The `instrument()` method can accept the argument of type `StoreDevtoolsConfig` defined in the node_modules/@ngrx/store-devtools/src/config.d.ts file. The next code listing shows how to add instrumentation that will allow monitoring of up to 25 recent actions and work in log-only mode in the production environment.

Listing 15.19 Adding instrumentation with two configuration options

```
StoreDevtoolsModule.instrument({
    maxAge: 25,
     logOnly: environment.production
 })
```

Retains the last 25 states in the browser extension

Restricts the browser extension to logOnly mode in production

You can also restrict some of the features of the Chrome Redux extension by providing the `features` argument to the `instrument()` method. For more details on configuring ngrx instrumentation and supported API, see http://mng.bz/3AXe, but here we'll show you some Chrome Redux extension screenshots to illustrate some of the features of ngrx store DevTools.

TIP If you run your app, but the Chrome Redux panel shows you a black window with the message "No store found," refresh the page in the browser.

Launching the app creates the initial state in the store. Figure 15.7 shows the screen after you launch the mediator app and enter `aaa` in the input field. The sequence of actions starts with two `init` actions that are dispatched internally by the packages `@ngrx/store` and `@ngrx/effects`, and you select the `@ngrx/store/init` action on the left and the State button at the top right. The state properties `searchQuery` and `searchResults` are empty. To see the app state after one of the search actions is dispatched, click this action.

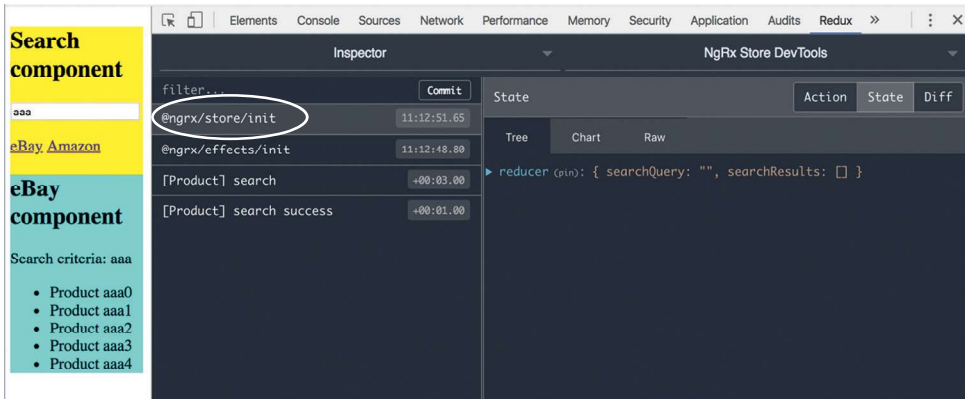


Figure 15.7 The store `init` action is selected.

Think of `init` actions as hooks that your app can subscribe to and implement some logic upon app launch—for example, you can check whether the user is logged in. If your app uses lazy-loaded modules that have their own reducers, you may also see the `@ngrx/store/update-reducer` action for each newly loaded module, and its reducer will be added to the collection of store reducers.

Figure 15.8 shows the screen after clicking the Action button at the top right, and shows the type and payload of the latest action:

1. The latest action is `"[Product] search success"`.
2. The Action tab is selected.
3. The action payload is stored in the state property `searchResults`.
4. The action type is `"[Product] search success"`.

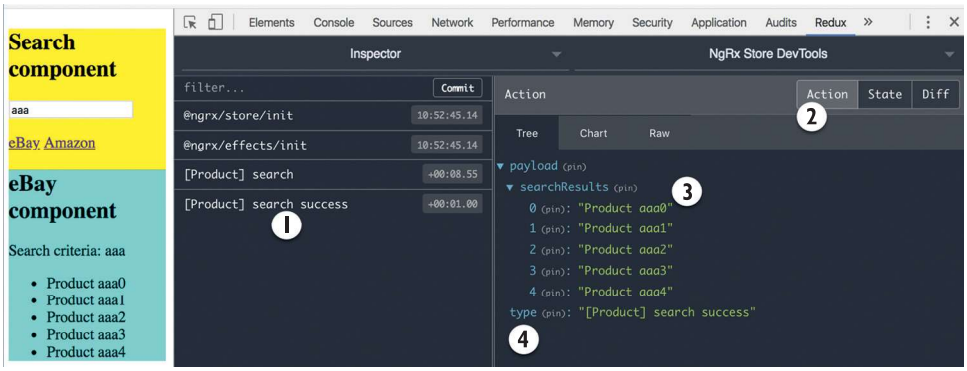


Figure 15.8 The Action tab view

TIP If your state object has many branches, by clicking `(pin)`, you can pin a certain slice of the state to the top while browsing actions.

As shown in figure 15.9, after clicking the State button, you can see the current values of your state variables `searchQuery` and `searchResults`:

1. The latest action is `"[Product] search success"`.
2. The State tab is selected.
3. The search criterion is stored in the state property `searchQuery`.
4. The search results are stored in the state property `searchResults`.

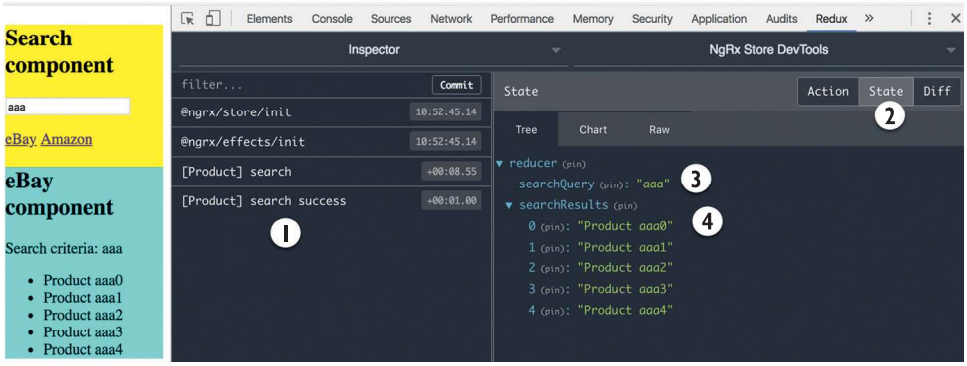


Figure 15.9 The State tab view

If the State tab shows the entire state object, clicking the Diff button shows what has changed as the result of the specific action. As shown in figure 15.10, if no action is selected, the Diff tab shows the state changes made by the latest action:

1. The latest action is `"[Product] search success"`.
2. The Diff tab is selected.
3. The content of the state property `searchResults` is different.

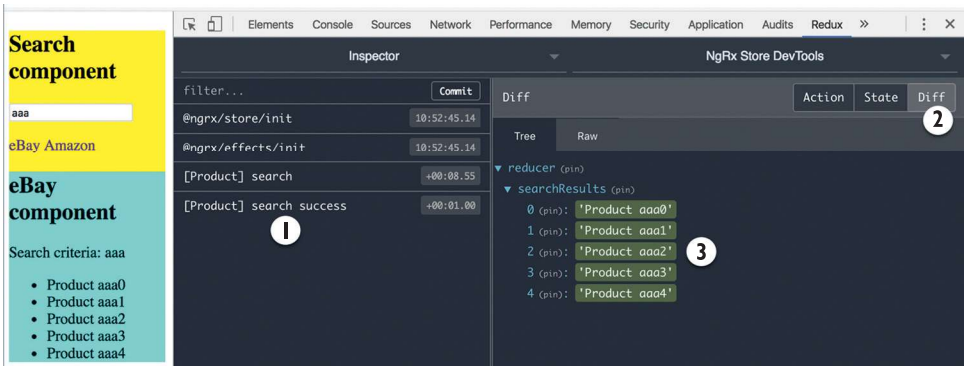


Figure 15.10 The Diff tab is selected.

While debugging an app, developers often need to re-create a certain state of the app, and one way to do that is to refresh the page and repeat user actions by clicking buttons, selecting list items, and so on. With Redux DevTools, you can travel back in time and re-create a certain state without refreshing the page—you can jump back to the state after a certain action occurred, or you can skip an action.

When you select an action, as shown in figure 15.11, you'll see the Jump and Skip buttons, and then clicking Skip will strike through the selected action, and your running app will reflect this change. The Sweep button will be displayed at the top, and clicking it removes this action from the list. The Jump button jumps you to a specific state of the app for a selected action. Redux DevTools will show you the state properties at the moment, and the UI of the app will be rerendered accordingly:

1. The Skip button for this action has been clicked.
2. The State tab is selected.
3. The search query is `aaabbb`.
4. The state property `searchResults` shows no results for the `aaabbb` products.
5. The Sweep button was not clicked.

We've shown you the main features of the ngrx store DevTools, but to understand this tool better, we encourage you to spend some time playing with it on your own.

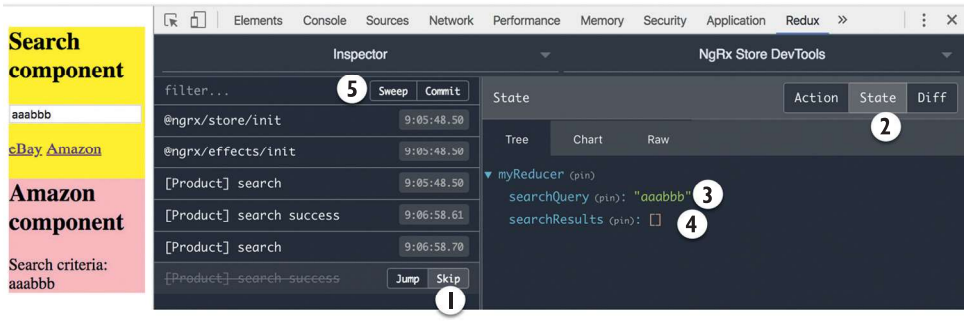


Figure 15.11 The [Product] search-success action is skipped

15.2.5 Monitoring the router state

When a user navigates an app, the router renders components, updates the URL, and passes parameters or query strings if need be. Behind the scenes, the router object represents the current state of the router, and the `@ngrx/router-store` package allows you to keep track of the router state in the ngrx store.

This package doesn't change the behavior of Angular `Router`, and you can continue using the `Router` API in components, but because the store should be a single source of truth, you may want to consider representing the router state there as well. At any given time, the ngrx store can give you access to such route properties as `url`, `params`, `queryParams`, and many others.

As with any other state properties, you'll need to add a reducer to the ngrx store, and the good news is that you don't need to implement it in your app because the `routerReducer` is defined in `@ngrx/router-store`. To add router state support, install this package first:

```
npm i @ngrx/router-store
```

After that, add `StoreRouterConnectingModule` to the `NgModule` decorator and add `routerReducer` to the list of reducers. `StoreRouterConnectingModule` holds the current router state. During navigation, before the route guards are invoked, the router store dispatches the action of type `ROUTER_NAVIGATION` that carries the `RouterStateSnapshot` object as its payload.

To get access to the `routerReducer` in your app, you need to perform two steps:

1. Give it a name by assigning a value to the property `StoreRouterConnectingModule.stateKey`.
2. Use the value from the previous step as the name of the `routerReducer`.

The following listing shows how the `StoreRouterConnectingModule` can be added to the app module. Here, you use `myRouterReducer` as the name of the `routerReducer`.

Listing 15.20 An app module fragment

```
import {StoreRouterConnectingModule, routerReducer}
  ➥from '@ngrx/router-store';
...
@NgModule({
  imports: [
    ...
      StoreModule.forRoot({myReducer: reducer,
                           myRouterReducer: routerReducer}),
    StoreRouterConnectingModule.forRoot({
      stateKey: 'myRouterReducer'
    })
  ]
    ...
})
export class AppModule { }
```

Adds routerReducer to the StoreModule

Stores the name of the reducer in the stateKey property

Now the state property `myRouterReducer` can be used to access the router state. The value of this property will be updated on each router navigation.

The app from section 15.2.3 didn't include router state monitoring, but the source code that comes with this chapter has another app called mediator-router, which does monitor router state. Run this app and open the Redux DevTools panel. Then navigate to the Amazon route and you'll see the `ROUTER_NAVIGATION` action and the `myRouterReducer` property in the app state object, as shown in figure 15.12.

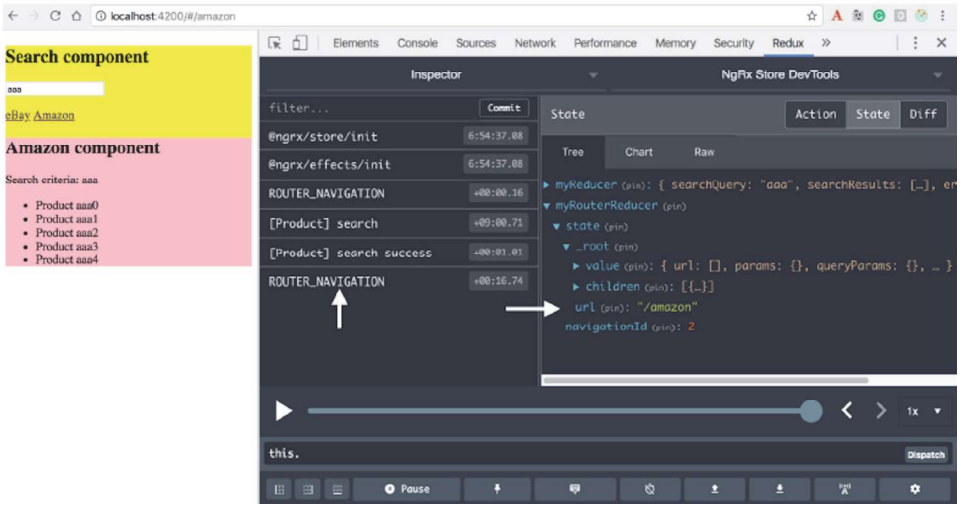


Figure 15.12 The `RouterStateSnapshot` object in Redux DevTools

TIP By clicking the down arrow on the bottom toolbar, a QA engineer can save the current state of the app and send it to a developer, who can load it into the Redux extension (up arrow) to reproduce the scenario reported as a bug.

Expand the nodes of `RouterStateSnapshot`; it has lots and lots of properties. This object is so big that it may even crash Redux DevTools. Typically, you need to monitor just a small number of router state properties, and this is where the router state serializer comes in handy.

To implement the serializer, define a type that will include only those properties of `RouterStateSnapshot` that you want to monitor. Then write a class that implements the `RouterStateSerializer` interface, and `@ngrx/router-store` will start using it. This interface requires you to implement the `serialize()` callback, where you should destructure the provided `RouterStateSnapshot` to extract only those properties you care about.

Some of the properties, like `url`, are available at the top level, and others, such as `queryParams`, are sitting under the `RouterStateSnapshot.root` property. The mediator-router project implements a router state serializer in the serializer.ts file, as shown in the following listing.

Listing 15.21 serializer.ts

```
import { RouterStateSerializer } from '@ngrx/router-store';
import {Params, RouterStateSnapshot} from '@angular/router';

interface MyRouterState {
   url: string;
  queryParams: Params;
}

export class MyRouterSerializer
        implements RouterStateSerializer<MyRouterState> {

  serialize(routerState: RouterStateSnapshot): MyRouterState {

    const {url, root: {queryParams}} = routerState;

    return {url, queryParams};
   }
}
```

Defines the router state properties you want to monitor

Creates a class that implements RouterStateSerializer

Uses destructuring to get only the properties you need

Returns an object with the properties url and queryParams

Now Redux DevTools will show only the values of `url` and `queryParams`. To get the value of the router state object, use the `select()` operator. The next listing shows how you do it in the mediator-router project.

Listing 15.22 app.component.ts

```
export class AppComponent {

  constructor(store: Store<any>) {
    store
```

```
      .select(state => state.myRouterReducer)
       .subscribe(routerState =>
           console.log('The router state: ', routerState));
  }
}
```

Extracts a router state slice

If you want to handle the router state action in your effects class, create an effect that handles the actions of type `ROUTER_NAVIGATION`. The following code listing from the effects.ts file from the mediator-router project shows how to do it in the effect.

Listing 15.23 A fragment of effects.ts

```
@Injectable()
export class SearchEffects {
...
  @Effect({ dispatch: false })
   logNavigation$ =
    this.actions$.pipe(
      ofType('ROUTER_NAVIGATION'),
       tap((action: any) => {
        console.log('The router action in effect:', action.payload);
       })
    );

  constructor(private actions$: Actions,
              private productService: ProductService) {}
}
```

This effect doesn't dispatch its own actions.

Listens to the ROUTER_NAVIGATION action

In some cases, you may want to arrange navigation inside the effects class. For this, shown in the following listing, you can keep using the router API without any help from ngrx.

Listing 15.24 Navigating in effects

```
@Effect({ dispatch: false })
navigateToAmazon$ =
  this.actions$.pipe(
    ofType('GOTO_AMAZON')
     tap((action: any) => {
      this.router.navigate('/amazon');
     })
  );
```

Listens to the GOTO_AMAZON action

Navigates to the /amazon route

This concludes our introduction to ngrx, but you'll see how you use it in the ngAuction app in the hands-on section of this chapter.

15.3 To ngrx or not to ngrx

Recently, one of our clients explained their needs for storing state. In their app, state is represented by a large object with nested arrays, and each array stores data rendered as

a chart. The app retrieves one of the arrays, performs some calculations, and renders a chart. In the future, they plan to add new charts and arrays.

The client asked whether using a singleton Angular service with `BehaviorSubject` would offer a less scalable solution than ngrx for this use case. He added that in ngrx, they could use separate arrays (state slices) with their reducers, which could make adding new arrays and charts easier because ngrx automatically creates one global state object from individual reducers.

Let's see if ngrx would help. First of all, if they needed lots of data to render the chart that doesn't use the data directly, it could make sense to move computations to the server to avoid keeping huge objects in memory and calculating numbers in the browser. But what if they still wanted to implement all the data crunching on the client?

With the Angular service approach, the object with nested arrays would grow and become less maintainable. In the case of separate reducers/arrays, it would be easier to add them to the state and reason about the state.

But with the ngrx approach, the state object would also grow, and they'd need to add more reducers and selectors to handle the growth. With the Angular service approach, they could either add more methods for getting the state slices or could split the singleton into multiple services—the main one would store the data, and separate services (one per chart) would get and process the data from the main service.

Both ngrx and service approaches can do the job and remain maintainable. If the app doesn't use ngrx yet, it wouldn't make sense to use ngrx just because of charts.

15.3.1 Comparing ngrx with Angular services

Okay, is there a use case where ngrx offers advantages over the Angular service approach? Let compare three main features of state management:

- A single source of truth could mean two things:
  a. There's only one copy of each set of data. This is easily achievable with an Angular service with `BehaviorSubject`.
  b. There's a single object that keeps all the app data. This is a unique feature of the Redux/ngrx approach that enables Redux DevTools. This can be a valuable feature for large apps with cross-module interaction and lots of shared data. Without a single state object, it would be nearly impossible. DevTools allows exporting/importing the entire state of the app if you need to reproduce a bug found by a user or a QA engineer. But in the real world, state changes trigger side effects and don't restore the app in exactly the same state.
- State can be modified only by a reducer, so you can easily locate and debug an issue related to state. But if you use `BehaviorSubject` to keep data in your Angular services, you can do this as well. Without `BehaviorSubject`, it would be hard to identify all assignments that can modify state, but with `BehaviorSubject`, there's a single place where you can put a breakpoint. Also, by applying the `map` operator to `BehaviorSubject`, you can handle all data modifications just like in a reducer.

- With ngrx and specific selectors, you can produce a derived state that combines data from different parts of the store object, plus it can be memoized. You can easily do this in an Angular service as well. Define a service that injects other services, aggregates their values with `combineLatest` or `withLatestFrom` operators, and then emits the "derived" state.

If you want all these features, it can be easier to start with ngrx because ngrx enforces them. Without enforced discipline, your singleton service that started with 30 lines of code can quickly turn into an unmaintainable monster with hundreds of lines. If you're not sure that best practices can be enforced in your team, go with ngrx, which offers a well-defined structure for your app.

TIP Instead of writing the code for creating a store, effects, actions, and so on, you can generate them using Angular CLI, but first install the ngrx blueprints (a.k.a. schematics). You can read about using `@ngrx/schematics` at http://mng.bz/7W30. If your project was generated by Angular CLI 6 or newer, you can add NGRX artifacts to it by using the following commands:

```
ng add @ngrx/store
ng add @ngrx/effects
```

15.3.2 State mutation problems

These issues do exist, but Angular has all you need to address them. In your projects, all Angular components use the change detection strategy `OnPush`. When you need to modify a component's data, create a new instance of an object bound to the component's `@Input`.

There are cases when using the default change detection strategy makes more sense. For example, you may need to create a dynamic form, and its content changes depending on the values entered in other form controls. Control values are exposed as observables (as `valueChanges`), and if your component uses `OnPush`, and all other component properties are RxJS `Subjects`, it makes the code overly complex to express the logic in terms of RxJS operators.

The code definitely can be complex, but often it doesn't have any benefits over disabling the `OnPush` strategy and mutating a component's state directly. Then don't use `OnPush`. On rare occasions, you can even manually trigger change detection with `ChangeDetectorRef`.

These techniques aren't a replacement for the immutable ngrx state, and they don't provide the same level of control over your data that ngrx does. But they help avoid problems caused by state mutation.

15.3.3 ngrx code is more difficult to read

Actions and reducers introduce indirection and can quickly complicate your code if used without care. A new hire would need to spend more time to become productive with your app since each component can't be understood in isolation. You may say the same is true for any Angular app that doesn't use ngrx, but we would disagree for two reasons.

First, components and services look pretty much the same in every Angular app, but every ngrx project has its own approach for implementing and organizing actions, reducers, store selectors, and effects. Actions can be defined as variables, classes, interfaces, and enums. They can be directly exposed as ES module members or grouped into classes. The same applies to reducers.

Second, supporting actions and reducers requires writing additional code that wouldn't exist in your app otherwise—it's not just moving the existing app code from components to ngrx entities. If your components are already complex, using ngrx could make the code difficult to read.

15.3.4 The learning curve

ngrx considerably steepens the learning curve. You need to learn how the packages `@ngrx/store` and `@ngrx/effects` work. You may also want to learn the `@ngrx/entity` package that helps normalize relational data. If you have experience working with relational databases, you know how easy it is to join data located in related tables. Using `@ngrx/entity` eliminates the need to create nested JavaScript objects (for example, a customer object with nested orders) and write complex reducers.

You also need to be comfortable with the RxJS library. It's not rocket science, but if you're already in the process of learning Angular, TypeScript, and all the tooling, it would be wiser to postpone adding libraries that you can live without.

15.3.5 Conclusion

Good libraries are those that allow you to write less code. Currently, ngrx requires you to write lots of additional code, but we hope that future versions of ngrx will be simpler to implement and understand. Meanwhile, keep an eye on a promising state management library called NGXS (see https://ngxs.gitbooks.io/ngxs), which doesn't require you to write as much code as ngrx and is built on TypeScript decorators. Another new project called ngrx-data (http://mng.bz/h6Nc) promises to support ngrx/Redux workflows with less coding.

Start with managing state using a singleton injectable service with `BehaviorSubject`. This approach may cover all your needs. Watch the video "When ngrx is an overkill" by Yakov Fain (www.youtube.com/watch?v=xLTIDs0CDCM), where he compares two small apps that manage state with and without ngrx. It's never too late to add ngrx to your app, so don't try to prematurely solve a problem you're not facing yet.

Now let's see how ngrx can be used in your ngAuction app.

15.4 Hands-on: Using ngrx in ngAuction

The ngAuction app that comes with this chapter uses ngrx for state management. It's modularized and has the root module `AppModule` and two feature modules, `HomeModule` and `ProductModule`. Since your feature modules are lazy loaded, we've added to each module the directory store, which in turn has its own subdirectories: actions, effects, and reducers, as shown in figure 15.13. Although this project has three directories named store, the running app will have a single store with merged states from each module.

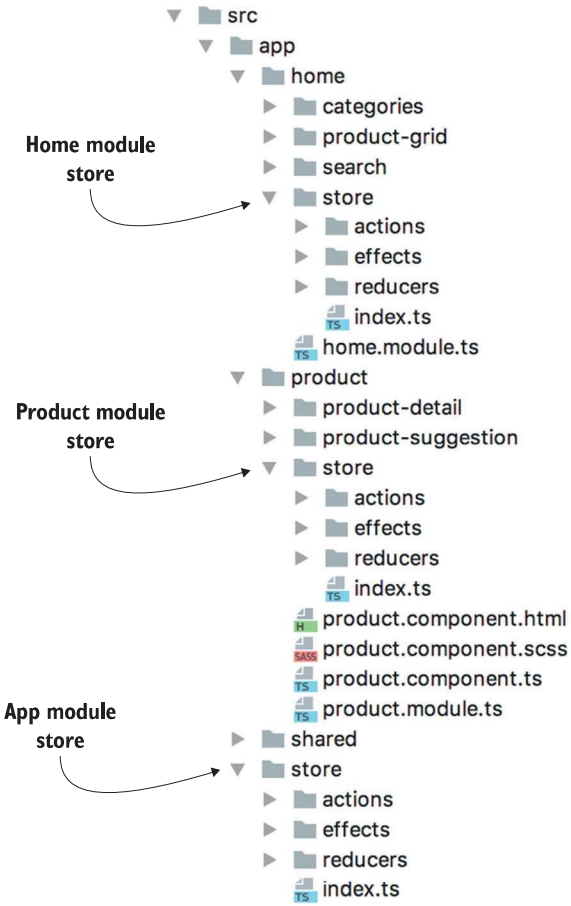


Figure 15.13 The state branches in the app and home modules

Inside of each folder—actions, effects, and reducers—we have separate files related to specific slices of state. For example, you can find a separate search.ts file that implements a respective piece of the search functionality in each of those folders.

App state can represent not only data (such as the latest search query or results), but also the state of the UI (such as the loading indicator is shown or hidden). You may also be interested to know the current state of the router and the URL displayed by the browser.

Figure 15.14 depicts the combined state of the running ngAuction. The names of reducers are shown in bold italic font, and arrows point at the state properties handled by each reducer. In particular, the `loading` property of the `products` reducer could represent the state of the progress indicator. We'll also add router support using the `router` reducer.

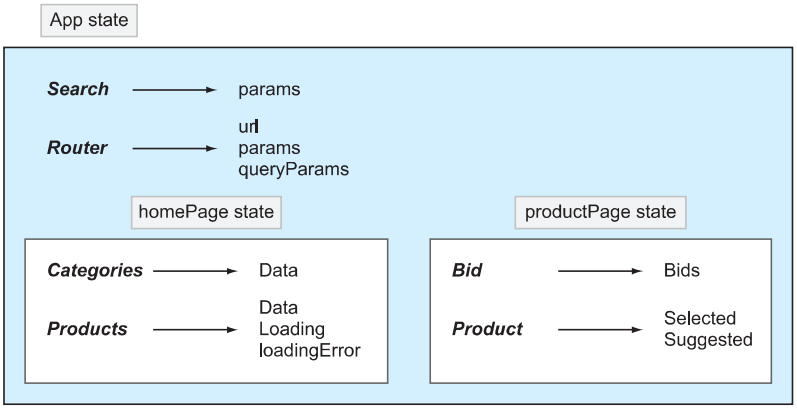


Figure 15.14 The combined state object of ngAuction

The router reducer is special in that you don't need to implement it in your app because it's defined in `@ngrx/router-store`, reviewed in the next section. Your ngAuction has the `@ngrx/router-store` package as a dependency in package.json.

Figure 15.15 shows a screenshot from Redux DevTools after ngAuction has launched and a user navigates to a specific product page. Note the router property there. The app state in figure 15.15 matches the state structure shown in figure 15.14:

- The State tab is selected.
- You see a `search` slice of state, a `router` slice of state, a `homePage` slice of state, and a `productPage` slice of state.

To run the ngAuction that comes with this chapter, you'll need to open two terminal windows, one for the client and one for server. Go to the server directory and run `npm install` there. Then, compile the code with the `tsc` command and start the server with the `node build/main` command. After that, open a separate Terminal window in the client directory and run the `npm install` command, followed by `ng serve`. We recommend you keep Redux DevTools open to monitor app state changes.

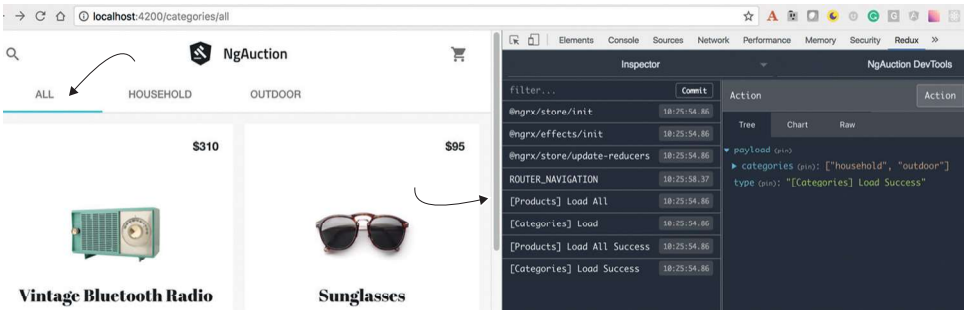


Figure 15.15 The ngAuction state in Redux DevTools

NOTE To keep the length of this section relatively short, we'll review just the code that implements state management in the home module and give you a brief overview of the router state. The state management of the product module is implemented in a similar fashion.

ngAuction uses four ngrx modules: `StoreModule`, `EffectsModule`, `StoreRouterConnectingModule`, and `StoreDevtoolsModule`, and the package for each of these modules is included in the dependencies section of package.json. Let's review the router-related code of the app module.

15.4.1 Adding the router state support to app module

When you select a product, the router navigates to the corresponding product view, and the URL changes accordingly—for example, http://localhost:4200/products/1. Selecting another product will change the router state, and you can bind these types of changes to the app state as well. The next listing shows the code fragments from app.module.ts focusing on the code related to router state support.

Listing 15.25 app.module.ts

```
import {EffectsModule} from '@ngrx/effects';
import {StoreRouterConnectingModule, routerReducer}
                from '@ngrx/router-store';
 import {StoreModule} from '@ngrx/store';
import {StoreDevtoolsModule} from '@ngrx/store-devtools';
import {environment} from '../environments/environment';
import {reducers,
         RouterEffects,
          SearchEffects} from './store';
...
@NgModule({
  imports: [
   ...
    StoreModule.forRoot({...reducers, router: routerReducer}),
     StoreRouterConnectingModule.forRoot({
       stateKey: 'router'
     }),
    StoreDevtoolsModule.instrument({
      name: 'ngAuction DevTools',
      logOnly: environment.production
    }),
    EffectsModule.forRoot([RouterEffects, SearchEffects]),
     ...
  ],
   ...
})
export class AppModule {
```

Imports the store module and the reducer for the router's state

Imports the router effects

Adds the routerReducer to the collection of app reducers

Adds the router state support

Names the router state property as router

RouterEffects listens to router events and dispatches ngrx actions handled by routerReducer.

TIP The next section provides more details about the line that loads reducers, while reviewing the code of the home module's index.ts file.

The name of the router state within the store is defined by the property name (for example, `router`) mapped to the router reducer. In your app, you'll use the default `routerReducer` and add it to the collection of app reducers:

```
StoreModule.forRoot({...reducers, router: routerReducer}),
```

The value of the `stateKey` property is used to find the router state within the store and connect it to Redux DevTools, so that time traveling during debugging works. The value assigned to `stateKey` (the `router`, in your case) must match the property name used in the map of reducers provided to the `forRoot()` method. To access a particular property of the router state, you can use the ngrx `select` operator on the object represented by the `router` variable.

Accessing the entire router state may crash Redux DevTools, which is why we created a custom router state serializer to keep in the store only the state properties you need. In the shared/services/router-state-serializer.service.ts file, we've implemented a serializer that returns an object containing only `url`, `params`, and `queryParams`. If we didn't implement this serializer, the router state shown in figure 15.14 would have lots of nested properties.

15.4.2 Managing state in the home module

When the home module is lazy loaded, its reducers are registered with the store, and its state object is merged with the root state. For this to happen, add the lines in the following listing to declare the store, reducer, and effects in the home.module.ts file

Listing 15.26 A fragment from home.module.ts

```
import {CategoriesEffects, ProductsEffects, reducers} from './store';
...
@NgModule({
  imports: [
  ...
   StoreModule.forFeature('homePage', reducers),
    EffectsModule.forFeature([ CategoriesEffects, ProductsEffects ])
   ]
```

Registers the reducers for the feature home module

Registers the effects for the feature home module

TIP The difference between the methods `forFeature()` and `forRoot()` is that the latter also sets up the required providers for services from the `StoreModule`.

The home module has reducers in the files store/reducers/products.ts and store/reducers/categories.ts. Note that you import reducers not from a specific file, but from the directory store, and you can guess that this directory has a file named index.ts that combines and reexports reducers from several files. You'll see the content of index.ts later in this section.

ACTIONS FOR PRODUCTS

In ngAuction, `CategoriesComponent` serves as a container of the home view, which renders the category tabs and the product grid on the home view. Figure 15.16 shows that `"[Products] Load All"` is the first action dispatched by the app. Then it dispatches `"[Categories] Load"`. When the data is loaded, two more actions are dispatched by the effects:`"[Products] Load All Success"` and `"[Categories] Load Success"`.

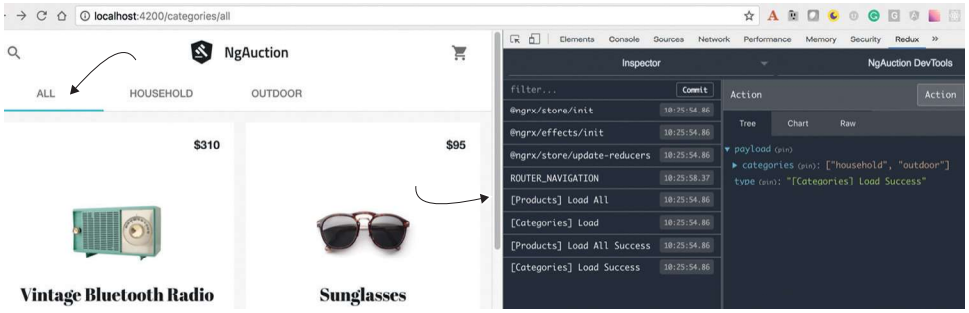


Figure 15.16 Loading products from all categories

Actions for categories are declared in the home/store/actions/categories.ts file, and actions for products in the home/store/actions/products.ts file. We'll review only the content of home/store/actions/products.ts; the categories actions are declared in a similar way.

In ngAuction, each file with actions usually consists of three logical sections:

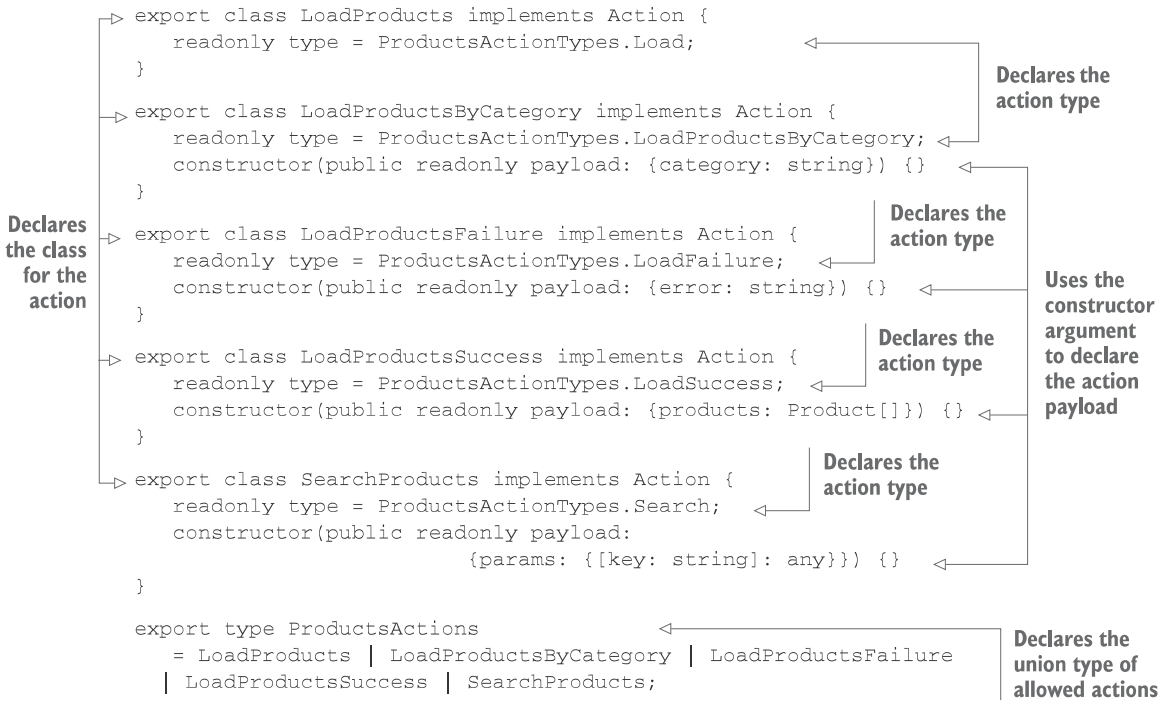
- The `enum` containing string constants defining the action types. You can read about TypeScript enums at http://mng.bz/sTmp.
- Classes for actions (one class per action) that implement the `Action` interface.
- The union type that combines all action classes. You'll use this type in reducers and effects, so the TypeScript compiler can check that the action types are correct, such as `ProductsActionTypes.Load`.

The next listing shows how actions are declared in the home/store/actions/products.ts file.

Listing 15.27 home/store/actions/products.ts

```
import {Action} from '@ngrx/store';
import {Product} from '../../../shared/services';

export enum ProductsActionTypes {
   Load = '[Products] Load All',
  Search = '[Products] Search',
  LoadFailure = '[Products] Load All Failure',
  LoadSuccess = '[Products] Load All Success',
  LoadProductsByCategory = '[Products] Load Products By Category'
}
```

Declares allowed action types as the enum of string constants

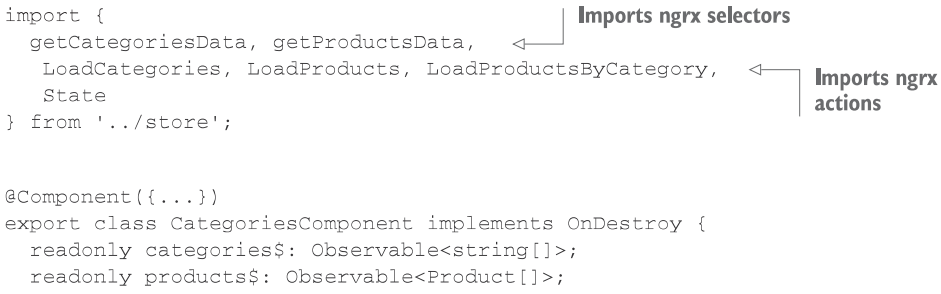


As you see, some of the action classes include only the action type, and some include the payload as well.

CATEGORIESCOMPONENT

The code of the `CategoriesComponent` has changed compared to the chapter 14 version. Fragments of the categories.component.ts file related to state management are shown in listing 15.28. In the constructor of the `CategoriesComponent`, you subscribe to the route parameters. When this component receives the category value, it either dispatches the action to load the products of all categories or only the selected one.

Listing 15.28 Fragments from categories.component.ts



```
  constructor(private route: ActivatedRoute,
              private store: Store<State>) {
    this.products$ = this.store.pipe(select(getProductsData));
    this.categories$ = this.store.pipe(
      select(getCategoriesData),
      map(categories => ['all', ...categories])
    );

    this.productsSubscription = this.route.params.subscribe(
      ({ category }) => this.getCategory(category)
    );
    this.store.dispatch(new LoadCategories());
  }

  private getCategory(category: string): void {
    return category.toLowerCase() === 'all'
      ? this.store.dispatch(new LoadProducts())
      : this.store.dispatch(new LoadProductsByCategory(
                  {category: category.toLowerCase()}));
  }
}
```

Injects the Store object

Subscribes to the categories to be rendered as tabs

Adds the all element to the array of category names

Loads the selected or all categories

Dispatches the action to load categories

Dispatches the action to load all products

Dispatches the action to load products by category

THE REDUCER FOR PRODUCTS

The home module has two reducers: one for products and one for categories. The reducers and selectors for products are shown in the following listing.

Listing 15.29 home/store/reducers/products.ts

```
import {Product} from '../../../shared/services';
import {ProductsActions, ProductsActionTypes} from '../actions';

export interface State {
   data: Product[];
  loading: boolean;
  loadingError?: string;
}

export const initialState: State = {
   data: [],
  loading: false
};

export function reducer(state = initialState, action: ProductsActions): State
      {
  switch (action.type) {
    case ProductsActionTypes.Load: {
       return {
        ...state,
        loading: true,
         loadingError: null
      };
    }
```

Declares the structure of the products state

The initial state has no products, and the loading flag is false.

Handles the Load action

Updates the loading flag because the loading begins

```
    case ProductsActionTypes.LoadSuccess: {
       return {
        ...state,
        data: action.payload.products,
         loading: false,
        loadingError: null
      };
    }

    case ProductsActionTypes.LoadFailure: {
       return {
        ...state,
        data: [],
         loading: false,
        loadingError: action.payload.error
       };
    }

    default: {
      return state;
    }
  }
}

export const getData = (state: State) => state.data;
export const getDataLoading = (state: State) => state.loading;
export const getDataLoadingError = (state: State) => state.loadingError;)
```

Handles the LoadSuccess action

The products are loaded—updates the state with data.

Handles the LoadFailure action

Removes products data, if any

Updates the error message

The accessors returning state properties

The state object for products has three properties: the array with products, the flag to control the loading indicator, and the text of the loading error, if any. When the reducer receives the action of type `Load`, it creates a new state object with an updated `loading` property that can be used by a component for showing a progress indicator.

If the `LoadSuccess` action has been dispatched, it indicates that the products were retrieved successfully. The reducer extracts them from the action's `payload` property and updates the state's `data` and `loading` properties. The `LoadFailure` action indicates that the products couldn't be retrieved, and the reducer removes the data (if any) from the state object, updates the error message, and turns off the `loading` flag.

At the end of the reducer script for products, you see three lines with the functions that know how to access the data in the products state object. You define these functions here to keep them where the `State` interface is declared. These accessors are used to create selectors defined in index.ts.

> **NOTE** The products reducer has no code that makes requests for data. Remember, the code communicating with external store parties is placed in the effects.

THE ROLE OF INDEX.TS IN HOME REDUCERS

In general, the files named index.ts are used for reexporting multiple members declared in separate files. This way, if another script needs such a member, you import this member from a directory without the need to know the full path to a specific file.

When reexporting members, you can give them new names and combine them into new types.

The home/store/reducers/index.ts file has the line `import * as fromProducts from './products';`, and to access exported members from the products.ts file, you can use the alias `fromProducts` as a reference—for example, `fromProducts.State` or `fromProducts.getData()`. With this in mind, let's review the code of the home/store/reducers/index.ts file in the following listing.

Listing 15.30 home/store/reducers/index.ts

```
import {createFeatureSelector, createSelector} from '@ngrx/store';
import * as fromRoot from '../../../store';
 import * as fromCategories from './categories';
 import * as fromProducts from './products';

export interface HomeState {
   categories: fromCategories.State;
  products: fromProducts.State;
}

export interface State extends fromRoot.State {
   homePage: HomeState;
 }

export const reducers = {
   categories: fromCategories.reducer,
  products: fromProducts.reducer
};

// The selectors for the home module

export const getHomeState =
               createFeatureSelector<HomeState>('homePage');
export const getProductsState =
               createSelector(getHomeState, state => state.products);
export const getProductsData =
               createSelector(getProductsState, fromProducts.getData);
export const getProductsDataLoading =
               createSelector(getProductsState, fromProducts.getDataLoading);
export const getProductsDataLoadingError =
               createSelector(getProductsState, fromProducts.getDataLoadingError);
export const getCategoriesState =
               createSelector(getHomeState, state => state.categories);
export const getCategoriesData =
               createSelector(getCategoriesState, fromCategories.getData);
```

Imports various exported members and gives them alias names

Combines states from categories and products

Declares the State type by extending it from the root State

Declares a feature named homePage to be used with StoreModule or createFeatureSelector()

Combines the reducers from categories and products

This script starts with creating descriptive alias names (for example, `fromRoot`), so it's easier to read the code knowing where a particular member is coming from. Then you declare a `HomeState` interface combining all the properties declared in the `State` interfaces in the reducers for both products and categories.

The app store includes one state object that can be a complex object containing multiple branches. Each of these branches is created by a module reducer. When an action is triggered on the store, it goes through each registered reducer and finds the

ones that have to handle this action. The reducer creates a new state and updates the corresponding branch of the global app state.

Here, you create a representation of the home module branch by declaring the `State` type that extends the `State` root and adding a new `homePage` property to it. You used this property in `createFeatureSelector()` and in listing 15.26 for registering the state object for the module home. When the combined app store is being formed, ngrx adds the `homePage` object to it.

The exported `reducers` member combines the reducers for products and categories. Now take another look at the app module in listing 15.25 which has the following line:

```
StoreModule.forRoot({...reducers, router: routerReducer})
```

Initially, the store finds and invokes each module reducer, which returns the corresponding state object. This is how the combined app state is created. The following code fragment from index.ts assigns the names `categories` and `products` to the respective slices of state:

```
export const reducers = {
  categories: fromCategories.reducer,
  products: fromProducts.reducer
};
```

At the end of the script, you declare and export all the selectors that can be used for retrieving slices of the home module state. Note that you use the state accessor functions declared in the respective reducer files.

EFFECTS FOR PRODUCTS

In the home module, effects are located in the files home/store/effects/categories.ts and home/store/effects/products.ts, and in the following listing, we review the code of the latter. The `ProductsEffects` class declares three effects: `loadProducts$`, `loadByCategory$`, and `searchProducts$`.

Listing 15.31 home/store/effects/products.ts

```
import {Injectable} from '@angular/core';
import {Actions, Effect, ofType} from '@ngrx/effects';
import {Action} from '@ngrx/store';
import {Observable, of} from 'rxjs';
import {catchError, map, switchMap} from 'rxjs/operators';

import {Product, ProductService} from '../../../shared/services';
import {LoadProductsByCategory, LoadProductsFailure,
    LoadProductsSuccess, ProductsActionTypes, SearchProducts} from '../actions';

@Injectable()
export class ProductsEffects {

  constructor(
    private readonly actions$: Actions,
    private readonly productService: ProductService) {}
```

```
@Effect()
loadProducts$: Observable<Action> = this.actions$
  .pipe(
    ofType(ProductsActionTypes.Load),
    switchMap(() => this.productService.getAll()),
    handleLoadedProducts()
  );

@Effect()
loadByCategory$: Observable<Action> = this.actions$
  .pipe(
    ofType<LoadProductsByCategory>(
➥ProductsActionTypes.LoadProductsByCategory),
    map(action => action.payload.category),
    switchMap(category => this.productService.getByCategory(category)),
    handleLoadedProducts()
  );

@Effect()
searchProducts: Observable<Action> = this.actions$
  .pipe(
    ofType(ProductsActionTypes.Search),
    map((action: SearchProducts) => action.payload.params),
    switchMap(params => this.productService.search(params)),
    handleLoadedProducts()
  );
}

const handleLoadedProducts = () =>
  (source: Observable<Product[]>) => source.pipe(
   map(products => new LoadProductsSuccess({products})),
   catchError(error => of(new LoadProductsFailure({ error })))
  );
```

Processes only Load actions

Tries to load all products

Dispatches either LoadProductsSuccess or LoadProductsFailure

Processes only LoadProductsByCategory actions

Extracts the category from the payload

Dispatches either LoadProductsSuccess or LoadProductsFailure

Tries to load products by provided category

Processes only Search actions

The function to dispatch either LoadProductsSuccess or LoadProductsFailure

Note the use of the `<LoadProductsByCategory>` type annotation in the `ofType` operator. This is one way of declaring the type of the action payload. Declaring the type explicitly (as in `map((action: SearchProducts))` is another way to do this.

Figure 15.17 shows the state after the action of type `LoadSuccess` is dispatched:

1. The search state is empty.
2. The router state shows the URL and parameters.
3. The categories state will be populated after the load success for categories is dispatched.
4. The state of products has data retrieved from the server.
5. The `loading` flag is `false`.
6. There are no errors.

As usual, the actions dispatched by effects will be handled by the reducer, which will update the state with the data or error message.

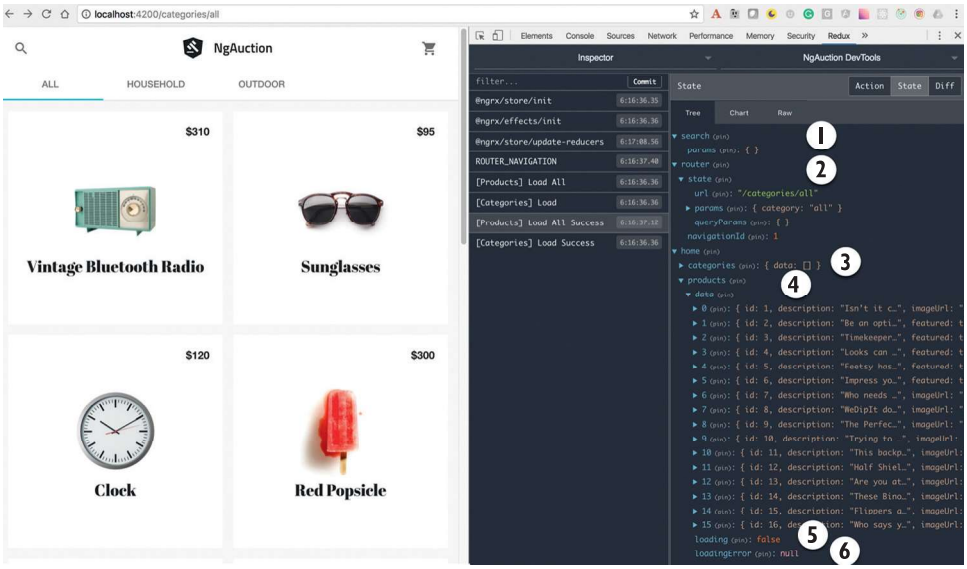


Figure 15.17 The state after the `LoadSuccess` action

15.4.3 Unit-testing ngrx reducers

Unit-testing state-related functionality is quite simple because only the reducer can change app state. Remember, a reducer is a pure function, which always returns the same output if the provided arguments are the same.

Because every action is represented by a class, you just need to instantiate the action object and invoke the corresponding reducer, providing the state and action objects to the reducer. After that, you assert that the state property under test has the expected value. For example, the home module has a reducer for products that defines the state object in the following listing.

Listing 15.32 The products slice in the home module state

```
export interface State {
  data: Product[];
   loading: boolean;
   loadingError?: string;
}
```

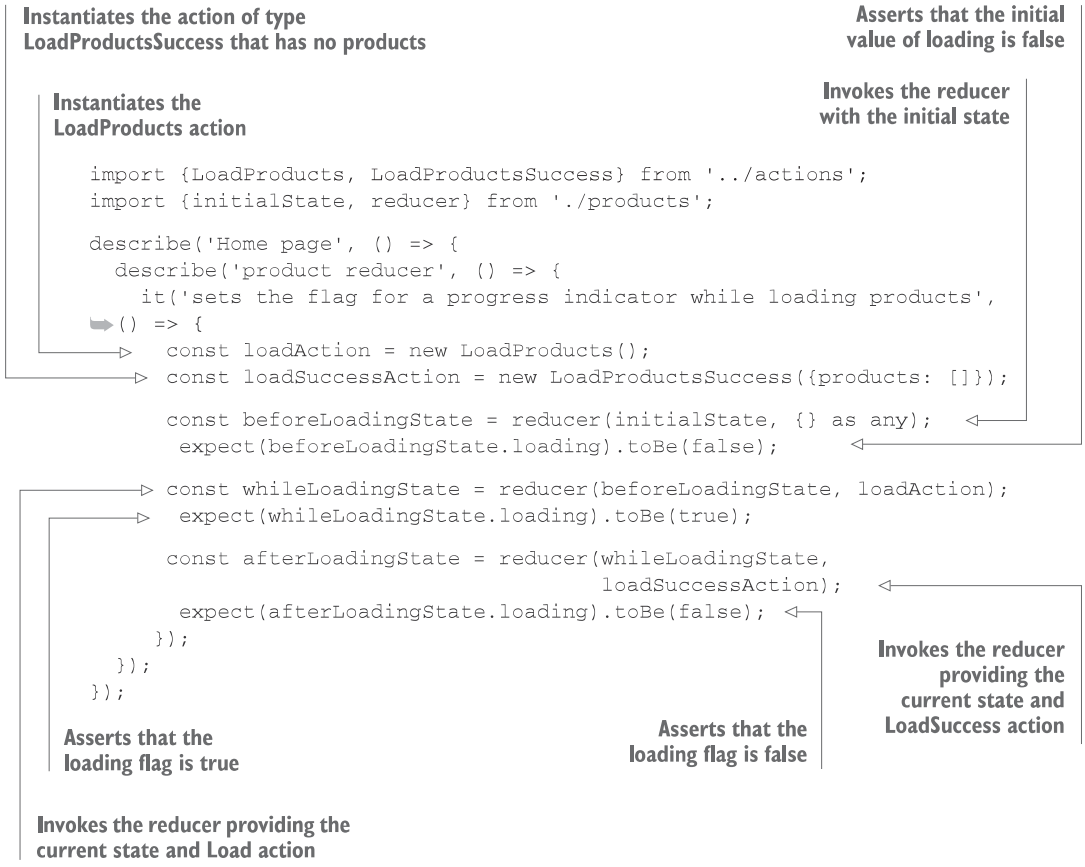
Current products are stored here.

If this flag is true, the UI should show a progress indicator.

If loading fails, the error message is stored here.

Let's review the code of the home/store/reducers/products.spec.ts file, shown in the following listing, which uses this state object and asserts that the `loading` flag is properly handled by the actions `LoadProducts` and `LoadProductsSuccess`.

Listing 15.33 home/store/reducers/products.spec.ts



When you invoke the reducer with the initial state, you provide an empty object and cast it to the type `any`, so regardless of the provided action, the reducer must return a valid state. Check the code of the reducer and note the `default` case in the `switch` statement there. Run the `ng test` command, and Karma will report that it executed successfully.

The listing 15.33 spec tests whether the reducer properly handles the `loading` property in the state object, without worrying about the action payload. But if you write a test for an action that has a payload, create a stub object with hardcoded data to simulate the payload and invoke the corresponding reducer.

This concludes our review of the ngrx code added to the home module of ngAuction. We encourage you to complete the code review of the product module on your own; its ngrx-related code is similar.

Summary

- The app state should be immutable.
- The app logic can be removed from components and placed in effects and services.
- A component's methods should only send commands (actions) and subscribe to data for further rendering.
- Although the ngrx learning curve is steep, using ngrx may result in better code organization, which is especially important in large apps.

Angular 6, 7, and beyond

The authors started working on the first edition of this book when Angular was in its early alpha versions. Every new Alpha, Beta, and Release Candidate was full of breaking changes. Writing the second edition was easier because Angular became a mature and feature-complete framework. New major releases come twice a year, and switching from one release to another isn't a difficult task. Every new release is tested against roughly 600 Angular apps used internally at Google to ensure backward compatibility.

We'd like to highlight some of the new features introduced in Angular 6 or planned for future releases:

- *Angular Elements*—Angular is a great choice for developing single-page applications, but creating a widget that can be added to an existing web page isn't a simple task. The Angular Elements package will allow you to create a self-bootstrapping Angular component that's hosted by a custom web element (see www.w3.org/TR/custom-elements/) that can be used in any HTML page.

  Simply put, you can define new DOM elements and register them with the browser. At the time of writing, all major browsers except Internet Explorer natively support custom elements; for IE, you should use polyfills.

  Say there's an existing web app built using JavaScript and jQuery. The developers of this app will be able to use Angular components (packaged as custom elements) in the pages of such an app. For example, if you build a price-quoter component, Angular Elements will generate a script that can be added to an HTML page, and your component can be used on an HTML page. Here's an example:

```
<price-quoter stockSymbol="IBM"></price-quoter>
```

  As you can guess, the `stockSymbol` is an `@Input` parameter of the Angular price-quoter component. And if this component emits custom events via its `@Output` properties, your web page can listen to them using the standard browser API `addEventListener()`.

  In our opinion, this killer feature will open many enterprise doors to the Angular framework. Angular Elements will be officially released in Angular 7.

- *Ivy renderer*—This is the codename of a new renderer that will make the size of an app smaller and the compilation faster. The size of the Hello World app is only 7 KB minified and 3 KB gzipped. This renderer will eliminate unused code while building bundles, as opposed to optimizing bundles, as is currently done. The Ivy renderer will be introduced in Angular 8.
- *Bazel and Closure Compiler*—Bazel is a fast build system used for nearly all software built at Google, including their 300+ apps written in Angular. Bazel makes it easier to publish Angular code that can be distributed as npm packages. The Closure Compiler is the bundling optimizer used to create JavaScript artifacts for nearly all Google web applications. The Closure Compiler consistently generates smaller bundles and does a better job of dead code elimination compared to Webpack and Rollup bundlers. By default, the Angular CLI project uses Webpack 4, which produces smaller bundles compared to its older versions.
- *Component Dev Kit (CDK)*—This package is already used by the Angular Material library, which offers 30+ UI components. Angular 6 introduces the tree component, which is good for displaying hierarchical data. The new flexible overlay component automatically resizes and positions itself based on viewport size. The badge component can show notification markers.

  What if you don't want to use Angular Material but want to build your own library of UI components and control page layouts? You can do that with CDK. CDK contains multiple subpackages, including overlay, layout, scrolling, table, and tree.

  For example, the CDK table deals with rows and columns but doesn't have its own styling. Although Angular Material adds styles to the CDK table, you can create your own according to your company guidelines. CDK supports responsive web design layout, eliminating the need for using libraries like Flex Layout, or learning CSS Grid. Angular 7 adds virtual scrolling for large lists of elements by rendering only the items that fit onscreen. Angular 7 also adds drag-and-drop support.
- *Angular CLI*—The .angular-cli.json file is renamed to angular.json, and its structure changes. The `ng update @angular/cli` command automatically converts existing .angular-cli.json into angular.json. The `ng update @angular/core` command updates the dependencies in your project's package.json to the latest version of Angular. If you need to upgrade your existing project to Angular 6, read Yakov Fain's blog, "How I migrated a dozen projects to Angular 6 and then 7," at http://mng.bz/qZwC. Upgrades from Angular 6 to 7 should not require code changes.

  The `ng new library <name>` command generates a project for creating a library instead of an app. This command will generate a library project with a build system and test infrastructure.

  The `ng add` command can add a package to your project, but in addition to what `npm install` does, it can also modify certain files in your project so you

don't need to do that manually. For example, the following command will install Angular Material, add a prebuilt theme to angular.json, add Material design icons to index.html, and add the `BrowserAnimationsModule` to the `@NgModule()` decorator:

```
ng add @angular/material
```

- *Schematics and* `ng update`—Angular CLI generates artifacts using a technology called *Schematics*, which uses code templates to generate various artifacts for your project. If you decide to create your own templates and have Angular CLI use them, Schematics will help you with this. The `ng update` command automatically updates your project dependencies and makes automated version fixes.

  With Schematics, you'll be able to create your own code transformations, similar to `ng update`. But you'll find some of the new prebuilt templates that come with Angular 6. For example, to generate all files for a `root-nav` component that already include the code of a sample Angular Material navigation bar, you can run the following command:

```
ng generate @angular/material:materialNav --name=root-nav
```

We're looking forward to all the new features that will make Angular even better!

With that, we'd like to thank you for reading our book. We hope you liked it and will leave positive feedback so that Manning will ask us to write a new edition in the future.