5
Implementing the Frontend with Angular Components

In the previous chapters of this book, we learned how easy it was to build a REST API with the Django framework, and we managed to get it done with truly little code and effort in very little time. As a result, we get the real benefit of efficiently concentrating on other important concerns of our application, like the application's business functionality or improving security, quality, and automation for CI/CD pipelines.

We are going to proceed with the second part of the book that deals with the frontend application, which is powered by different technologies from those that powered the backend.

As you already know from *Chapter 1*, the frontend is going to be implemented as a **single page application** (**SPA**) with Google's Angular framework.

In this chapter, we will focus on the main Angular **components**, which consist of the **forms** the user will interact with.

In this chapter we are going to cover the following main topics:

- Installing the development environment
- Installing and configuring the project
- Creating Angular components

Technical requirements

The `README.md` is the most important file in the project as it is the starting point for any new developer joining and should contain precious information like the installation instructions for the application as well as the project's best practices used in coding and management of the software lifecycle. Should contain information like naming conventions in source code, naming conventions in **branches and commits**, **merge requests** (**MR**), **code reviews**, **Definition of Done** (**DoD**) for tasks, etc.

To avoid the file becoming too big, it may contain links to further information in the company's content management sites like Atlassian Confluence. The README file is written using the Markdown syntax, therefore the file extension `.md`; this syntax is commonly used for documentation in content management tools.

You should be able to install the application that contains the source code used in this book by following the instructions in the `README.md` file without any inconsistency or ambiguity. It should be clear to understand and follow when it is rigorous and is up to date.

If you join a project where the README file is empty or missing, that speaks very poorly about the project itself and most certainly means that the quality of the application is going to be similar, that is extremely poor. This file is the welcome introduction for someone joining a project so is the first impression they will get of it.

Specific hardware/software requirements

Software

- Git
- Angular CLI 16.1.6
- Node.js 20.5.0
- Package manager: npm 9.8.0

This project has been created with Angular CLI version 16.1.6 (https://github.com/angular/angular-cli) and Node 20.5.0 on a machine with Ubuntu OS version 22.04.1

Browser support

The clients will use the Angular frontend application through a browser. Here is a list of compatible browsers for Angular:

- IE 9+
- Microsoft Edge
- Mozilla Firefox 22+

- Chrome 17+
- Opera 12+
- Safari 5+

Installing the development environment

Before starting to build our frontend application we will need a set of tools installed on your computer as prerequisites (see *Technical requirements* section). We can install them by following the instructions contained in the `README.md` file in the frontend main folder of the Git repository (https://github.com/PacktPublishing/Full-Stack-Web-Development-with-Django-and-Angular/tree/main/frontend/README.md).

I have installed the environment on a machine with Ubuntu OS version 22.04.1, but if you have a different operating system on your computer, you can find links for installation in different operating systems in the README file.

To proceed with the following installation steps, you need to open a terminal in your computer and execute the required commands that are explained as follows.

Git installation

First you will need to install Git to be able to clone the Git repository for the application. Execute the following commands:

```
$ sudo apt install git-all

$ git –version
```

If Git is installed, you should see the version of your installation:

```
git version 2.34.1
```

For installing Git on other operating systems, see the following link:

https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

Angular installation

Next, you will need to install Node.js version 20 and Angular CLI version 16 to work with the Angular framework. Execute the following commands for installation:

- Installing Node.js:

```
$ brew install node@20

$ echo 'export
PATH="/opt/homebrew/opt/node@20/bin:$PATH"' >>
~/.zshrc
```

For installing Node.js on other operating systems, see the following link:
https://nodejs.org/en/download/package-manager

Check the installation with the following command:

```
$ node -v
```

- Installing Angular CLI (Command Line Interface)

```
https://nodejs.org/en/download/package-manager
```

Check the installation with the following command:

```
$ ng version
```

If Angular is installed, you should see the version of your installation:

```
Angular CLI: 16.1.6

Node: 20.5.0

Package Manager: npm 9.8.0

OS: linux x64
```

Once you have installed all the tools required by the frontend application on your computer, you can proceed to the next step, which consists of installing the example frontend application for this tutorial on your computer.

Installing and configuring the project

Now that you have the required tools installed in your development environment you can go ahead and clone the git repository (in case you have not cloned it before as was required in *Chapter 2* to build the backend application) that holds the source code of this book in a local folder of your choice on your computer. You can use the `git`

`clone` command in a terminal or with your preferred IDE. I recommend using Visual Studio Code as the IDE because it has good support for both Django and Angular:

```
$ git clone https://github.com/PacktPublishing/Full-Stack-Web-Development-with-Django-and-Angular.git
```

Once you have cloned the git repository in your computer, you can find the `README.md` file in the main folder of the project where you can see the same installation instructions as in this book.

Now you have all the necessary tools and source code to explore and play with the provided code, to run it and execute the automated tests.

You can also find an optional section in the README file where you can see the steps to follow when initializing the Django application from scratch instead of downloading the provided application.

To install the project in your IDE, open a console or terminal, place yourself in the root directory for the frontend (`Full-Stack-Web-Development-with-Django-and-Angular/frontend/angularpackt-ui`) and run the following command:

```
$ npm install
```

This will install all the required libraries and dependencies in your local environment to make the frontend application work.

In case you´d like to start generating the initial structure for your own project, use the Angular CLI running the following command:

```
$ ng new angularpackt-ui --prefix apui --routing
```

Remember to replace the parameter `angularpackt-ui` with your own choice for your application name.

The `--prefix` option is used to differentiate our own directives from the Angular directives. In our example, our own directives will be inside the `<apui> </apui>` tags.

The `--routing` option is used to additionally generate an Angular router.

The process of automatically generating the application will ask you to select the stylesheet format:

```
$ Which stylesheet format would you like to use?
```

You can select one of the options with the cursor going up and down.

When the Angular CLI finishes the application generation, you will get a structure of folders and files like that shown in the figure below:

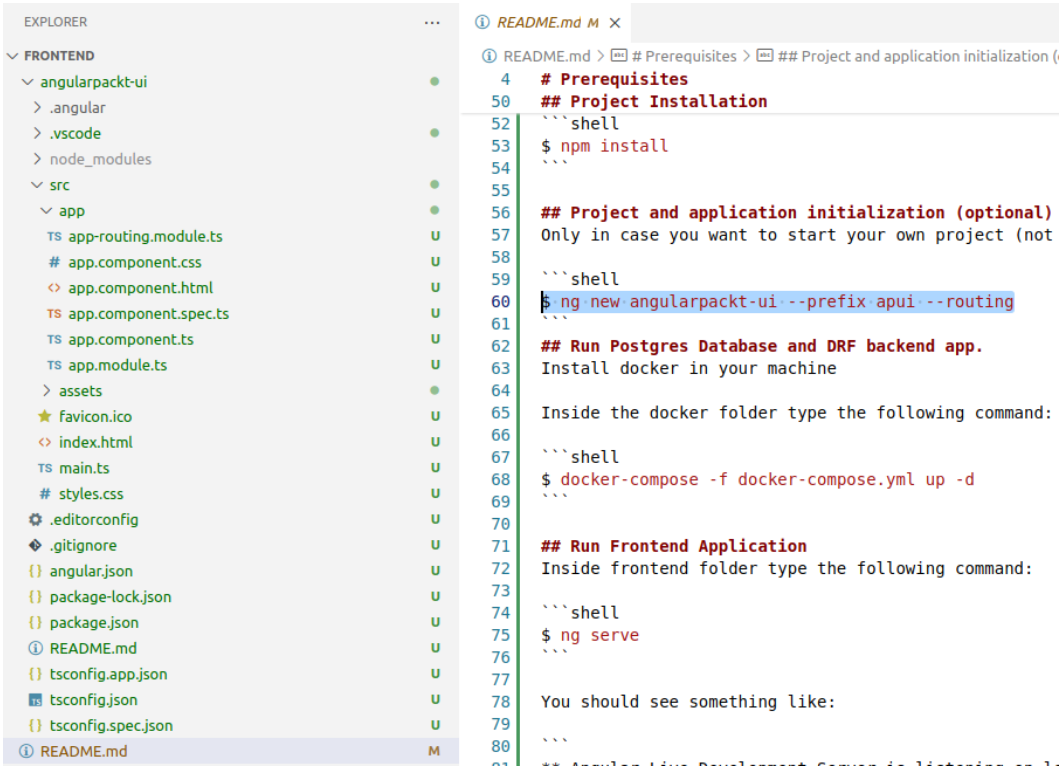


Figure 5.1 – Application initial skeleton and main files

Here is a description of the main files and structure of the generated project:

- `README.md`: Documentation for the workspace.
- `angular.json`: CLI configuration defaults for all projects in the workspace, including configuration options for build, serve, and test tools that the CLI uses.
- `package.json`: Configures npm package dependencies that are available for all projects in the workspace.
- `package.lock.json`: Provides version information for all packages that are installed in the `node_modules` folder by the npm client.

- `node_modules` folder: Folder where npm packages for the entire workspace are installed.
- `tsconfig.json`: The base TypeScript configuration for projects in the workspace.
- `tsconfig.app.json`: Application-specific TypeScript configuration, including Angular compiler options.
- `tsconfig.spec`: TypeScript configuration for application tests.
- `src` folder: Contains the source files for the root-level application folders.
- `main.ts`: The main entry point for your application.
- `Index.html`: The main HTML page that is served when someone visits your page.
- `styles.css`: Global CSS styles applied to the entire application.
- `favicon.ico`: An icon to use for this application in the toolbar bar.
- `src/app` folder: Contains your project logic and data. Angular components, templates, and styles go there.
- `src/app/app-routing.module.ts` Defines the application routing configuration.
- `src/app/app.module.ts`: Defines the root module, named `AppModule`, that tells Angular how to assemble the application.
- `src/app/app.component.ts`: Defines the application's root component named `AppComponent`.
- `src/app/app.component.spec.ts`: Defines an automated test for `AppComponent`.
- `src/app/app.component.html`: Defines the HTML template associated with `AppComponent` and is the single page or unique container that renders the application content **SPA**.
- `src/app/app.component.css`: Defines the CSS stylesheet for `AppComponent`.

- `src/assets` folder: Defines a folder where to include hardcoded or fake data used for testing and development when some dependencies or resources are not yet available.

That was a good list of files for the initially generated application. You will get used to them when you start developing and adding new components to the application.

Running the application

Now that you have a fully working Angular application you can start it with the following command:

```
$ ng serve


Full-Stack-Web-Development-with-Django-and-
Angular/frontend/angularpackt-ui$ ng serve
```

The `ng serve` command will start the application and if everything goes well after compilation you will see an output like this:

```

** Angular Live Development Server is listening on
localhost:4200, open your browser on http://localhost:4200/
**
```

Now you can open your preferred browser and point to this URL to see the initial page for the frontend as shown in the figure below:

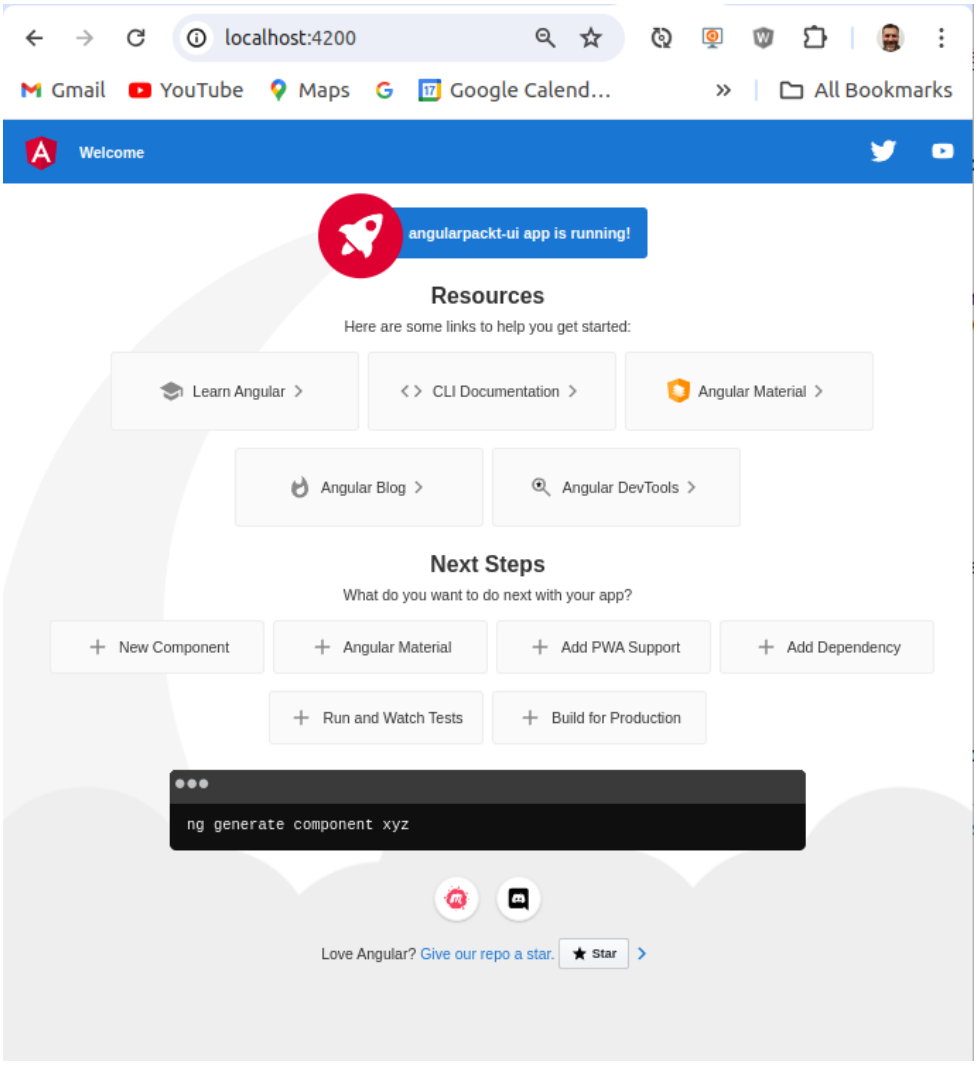


Figure 5.2 – Application main page

You can see the contents contained inside the `app.component.html` page generated automatically. This page is the container for the SPA. That means that Angular only serves this single page and the content inside the dynamic section of the page is updated when the route of the application changes, thus showing the content of other components.

The HTML dynamic section inside `app.component.html` is recognized with this tag:

```
<router-outlet></router-outlet>
```

That means that the content for a component that is accessed in a different route is automatically added inside this section in the `app.component.html` file.

SPA

This design is commonly used by many JavaScript frameworks, where a unique page is shown by the server, but the content dynamically changes as the route or URL that accesses the application changes

//

Running the tests

When we generated the initial project structure, the `AppComponent` was automatically created by the Angular CLI for us in the `app.component.ts` file. The source code for this component also included some automated tests. The file containing the tests is `app.component.spec.ts`.

If you have noticed, the name of the test file is the same as the component but with the `.spec` (specification) suffix before the `.ts` (TypeScript) suffix. This is a convention for all the tests. The tests and the components are both written in TypeScript language.

The tests are written in TypeScript and use the **Jasmine** (BDD) testing framework, and a test runner called **Karma**.

You can run the tests with the following command:

```
$ ng test
```

You will see an output like this in the terminal:

```
03 07 2024 15:32:21.934:WARN [karma]: No captured browser,
open http://localhost:9876/

03 07 2024 15:32:21.941:INFO [karma-server]: Karma v6.4.3
server started at http://localhost:9876/

03 07 2024 15:32:21.941:INFO [launcher]: Launching browsers
Chrome with concurrency unlimited

⁘ Generating browser application bundles...03 07 2024
15:32:21.962:INFO [launcher]: Starting browser Chrome

✓ Browser application bundle generation complete.

03 07 2024 15:32:22.414:INFO [Chrome 125.0.0.0 (Linux
x86_64)]: Connected on socket uyGjQIUaEvrHQKKhAAAB with id
80209664

Chrome 125.0.0.0 (Linux x86_64): Executed 3 of 3 SUCCESS
(0.07 secs / 0.063 secs)

TOTAL: 3 SUCCESS
```

//

Additionally, when the tests run, the Karma runner will automatically open a browser window in the frontend on a random port, so we can graphically see the result of the tests. You can see how this test report looks like in the figure below.

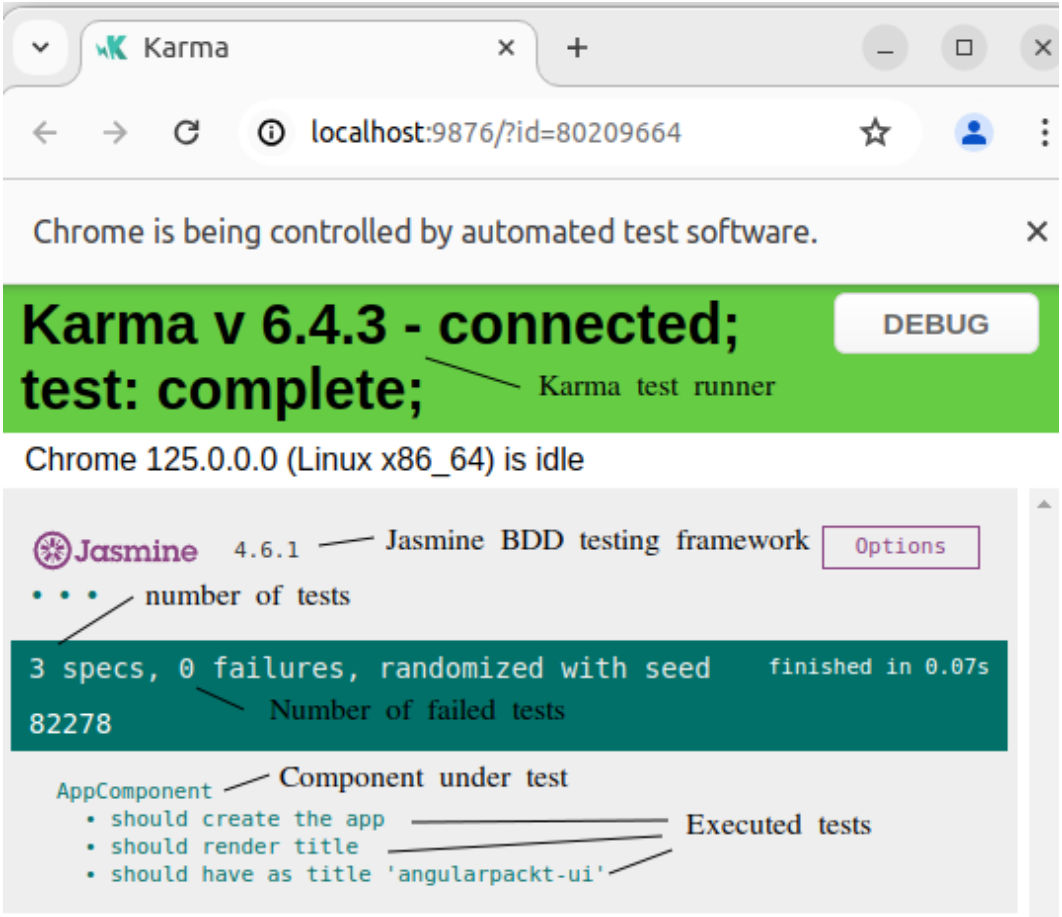


Figure 5.3 – Running automated tests with Jasmine and Karma

As you can see in the figure above, three tests for the `AppComponent` are run with no failures, so the report color is green. If any test fails, the color of the report will be red, and it will show the details of the failed tests.

But we will explain the topic of testing the frontend with more detail in *Chapter 9*.

Just do not forget to run the tests again whenever you make changes in the application and add more tests in case you add new code.

In case some test fails, fix it right now before it is too late. It is critical to keep the test report always green to keep your application healthy.

//

Creating Angular components

At this point we have everything we need to build our application and add new components and features to it.

Components are the main parts of an Angular project, as they build the HTML template containing the graphical representation, the source code containing the logic, and the stylesheet containing the look and feel for each of the domain models. So, they are the cornerstone to implementing dynamic content in an SPA.

As explained previously, the application contains a single page served by the `app.component.html` page and its content is automatically updated when we navigate through the different routes of the application.

If you need to check and play with Angular code before implementing it in your application, you can use this cool link available for learning purposes: https://angular.dev/playground

Let's start by creating a simple component for managing the products we created in the Django backend in previous chapters. As a reminder, `Products` was part of the data model for our backend application in this tutorial. We created a REST API for managing products with any external client, in this example, our Angular frontend.

Here, we will just generate the component files and tests, but we will leave the integration with the REST API for the next chapter.

To generate the structure of an Angular component with the Angular CLI, first we will manually create a new folder called `components` under the `src/app/` folder. The purpose of this is to differentiate the components we add for our application implementation from the `AppComponent` that was automatically created initially when we generated the application.

Once you have created the new `components` folder, place your terminal in the `/src/app/components/` and execute the following command:

```
$ ng generate component product
```

Executing this command will automatically generate all the necessary changes for us, and will show the following output in the terminal:

```
Full-Stack-Web-Development-with-Django-and-
Angular/frontend/angularpackt-ui/src/app/components$ ng
generate component product

CREATE src/app/components/product/product.component.css (0
bytes)

CREATE src/app/components/product/product.component.html
(22 bytes)

CREATE src/app/components/product/product.component.spec.ts
(566 bytes)

CREATE src/app/components/product/product.component.ts (207
bytes)

UPDATE src/app/app.module.ts (490 bytes)
```

As you can see in this output log, four new files are automatically generated for the new `ProductComponent`. And these new files are created under the automatically generated `product` folder, so we can organize the files associated for each component in its own folder.

Every time you create a new component, the same four file types are created as when the initial `AppComponent` was generated. Here is an explanation for each of them (the name of the component inside square brackets is generic and corresponds to the name of the component you generate):

- `[NewComponent].component.ts`: The TypeScript code that will handle the form logic for the component.
- `[NewComponent].component.html`: The HTML form or UI (graphical) representation for the component.

  `[NewComponent].component.css`: The stylesheet for the look and feel of the form for the component.
- `[NewComponent].component.spec.ts`: The automated test for the code logic inside the `[newComponent].component.ts` file.

Let's have a look inside the generated files for the new `ProductComponent`:

- `product.component.css`: This file is empty because it does not contain any customizations for the styles in the page. So, the default styles will be applied by Angular.
- `product.component.html`: This file contains a minimalistic version for the contents of the HTML form for the component:

```
<h1><p>product works!</p></h1>
```

- `product.component.ts`: This file contains the TypeScript code that powers the dynamic part in the component and links the TypeScript file (`.ts`) with the style (`.css`) and HTML page (`.html`) files:

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


@Component({
selector: 'apui-product',
templateUrl: './product.component.html',
styleUrls: ['./product.component.css']
})


export class ProductComponent {
}
```

- `product.component.spec.ts`: This file contains the Jasmine-automated test written in TypeScript code for the component. We will dig into the details for testing components in *Chapter 9*:

```
import { ComponentFixture, TestBed } from
'@angular/core/testing';
```

```
import { ProductComponent } from
'./product.component';


describe('ProductComponent', () => {
let component: ProductComponent;
let fixture: ComponentFixture<ProductComponent>;


beforeEach(() => {
TestBed.configureTestingModule({
declarations: [ProductComponent]
});
fixture =
     TestBed.createComponent(ProductComponent);
component = fixture.componentInstance;
fixture.detectChanges();
});


it('should create', () => {
expect(component).toBeTruthy();
});
});
```

As you can see in the previous output log, the `app.module.ts` file was also changed to include the new component in the application. The new `ProductComponent` was included in the `declarations` array inside the `@NgModule` block:

```
@NgModule({
declarations: [
```

```
AppComponent,
ProductComponent
],
imports: [
BrowserModule,
AppRoutingModule
],
providers: [],
bootstrap: [AppComponent]
})
```

If you run the test suite again, you will see that the test report is still green but contains an additional test for the new component. See figure below:

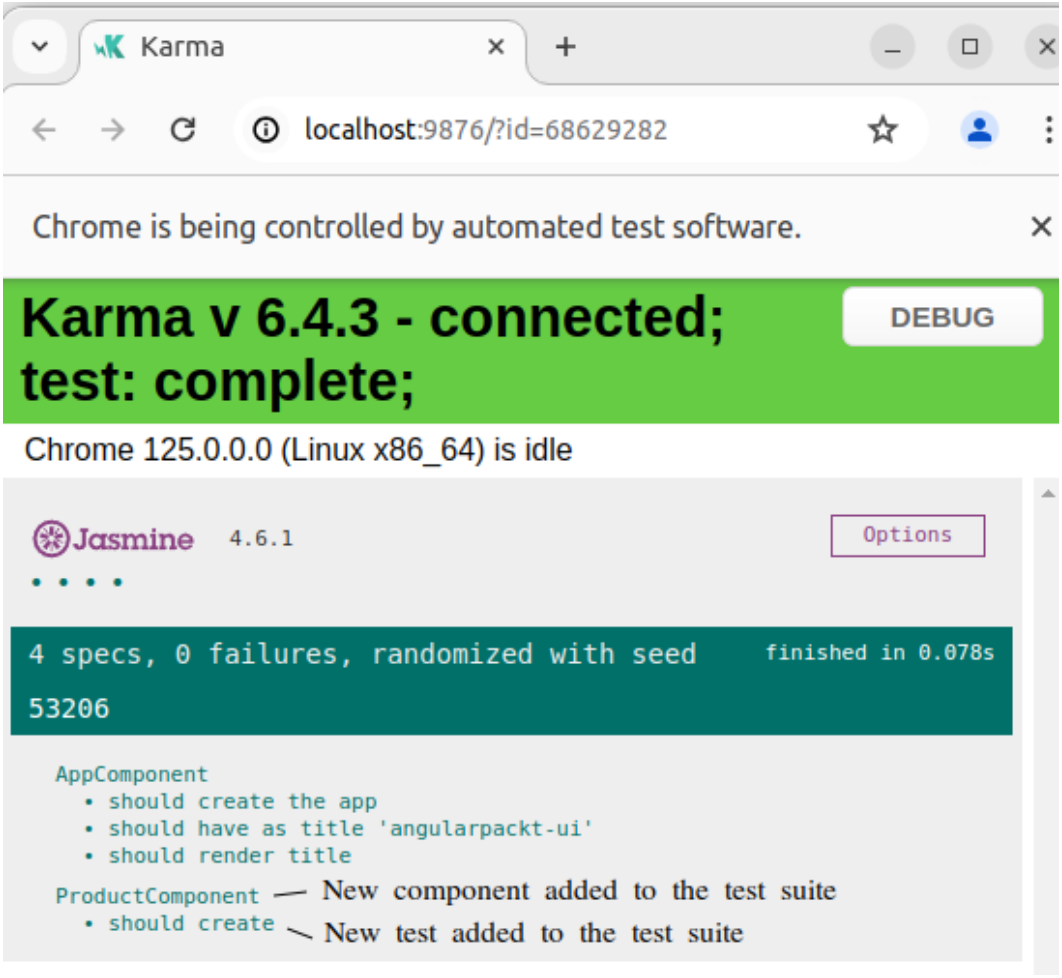


Figure 5.4 – Adding new component to the test suite

Next, we are going to learn how to add a route to access the new component from the UI.

Creating the route for the new component

We need to create a route for the new `ProductComponent` we have created in the previous section to be accessible from the UI in a new route (URL).

To create the new route we need to add a new path in the `Routes` array inside the `app-routing.module.ts` file as shown in the figure below.

Add this line:

```
{ path: 'product', component: ProductComponent }
```

Figure 5.5 – Adding the route for the new component

If you now open the new route in the UI with your preferred browser pointing to the `http://localhost:4200/product` URL, you should see the contents of the `ProductComponent` html page included in the main html file (`app.component.html`):

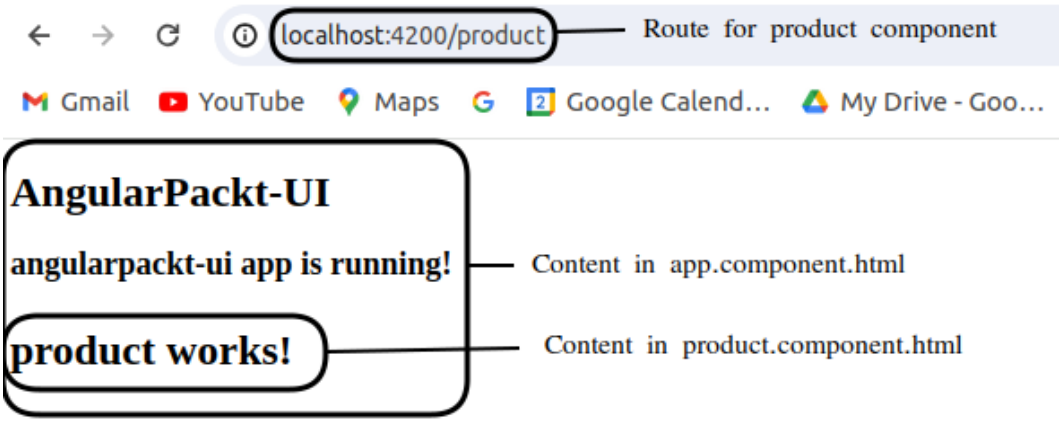


Figure 5.6 – Accessing the new route in the UI

As you can see in the preceding figure, the SPA dynamic section is updated with the contents of the `Product` component when the route changes.

Next, we are going to learn how to add a form to manually manage records for our new component from the UI.

Creating a form for a component

Next, we need to create form pages for managing records manually with the UI. We should be able to perform common REST operations like create, read, update, delete, as well as show a paginated list of records found after searching by important fields in the model.

In Angular you can use basically two different technologies from the Forms API for implementing forms: **reactive forms** and **template-driven forms**.

Reactive Angular forms API
With this solution approach, you explicitly create the model object in TypeScript code and then link the HTML template elements to that model's properties using special directives. This approach can suit the most complex forms; therefore, it is what we are going to use in this tutorial because it is the most powerful solution.

Template-driven Angular forms API
This solution approach is only programed in the component template using directives, and the model is created implicitly by Angular. It is limited just to HTML syntax and is suitable only for simple forms.

Both form APIs need to be explicitly activated in the application before you start using them.

You can activate the reactive-forms API by adding the following line in the `@NgModule` imports array inside the `app.module.ts` file:

```
ReactiveFormsModule,
```

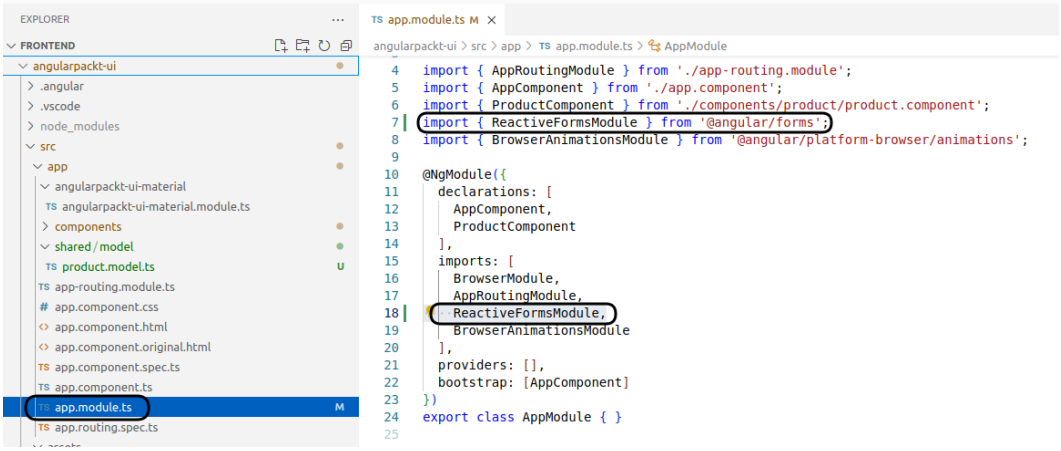


Figure 5.7 – Enabling Reactive forms API

You will also need one enhancement for Angular: the Material design components for Angular.

Angular Material

The Angular Material library is provided by the same Angular team to improve the look and feel of your forms with standard components that otherwise will require a lot of effort to implement on your own and that will give your application a modern and powerful look and feel, improving the user experience.

To install Angular Material in your application you need to execute the following command:

```
$ ng add @angular/material
```

Here is the output after executing the command:

```
Full-Stack-Web-Development-with-Django-and-
Angular/frontend/angularpackt-ui$ ng add @angular/material

i Using package manager: npm

✓ Found compatible package version:
@angular/material@16.2.14.

✓ Package information loaded.
```

```
The package @angular/material@16.2.14 will be installed and
executed.

Would you like to proceed? Yes

✓ Packages successfully installed.

? Choose a prebuilt theme name, or "custom" for a custom
theme: Indigo/Pink        [ Preview:
https://material.angular.io?theme=indigo-pink ]

? Set up global Angular Material typography styles? Yes

? Include the Angular animations module? Include and enable
animations

UPDATE package.json (1114 bytes)

✓ Packages installed successfully.

UPDATE src/app/app.module.ts (678 bytes)

UPDATE angular.json (2919 bytes)

UPDATE src/index.html (584 bytes)

UPDATE src/styles.css (181 bytes)
```

As you can see in the previous log, after you answer the questions about the custom theme, global styles, and animations, it will automatically update four files in your application.

As the last step for the installation, you will need to add material as a module to the application by adding a new file called `material-module.ts` inside the `src/app/` folder. And you will need to import this new module into the `app.module.ts` file.

You can get full details about this installation in the Angular Material web page located in the following URL:

https://material.angular.io/guide/getting-started

Implementing the component form template

Next, we are going to implement the HTML form template for the `Product` component in the `product.component.html` file using the reactive-forms API and Angular Material as shown in the figure below:

product.component.html M

angularpackt-ui > src > app > components > product > product.component.html > form > div.form-container > button

```
<form [formGroup]="productForm" (ngSubmit)="onSubmit()">
    <div class="form-container">
        <mat-form-field appearance="outline">
            <mat-label for="name" class="form-control">Name: </mat-label>
            <mat-hint>The product name  </mat-hint>
            <input matInput placeHolder="Product Name" data-testid="name" type="text"
                formControlName="name" required uppercase>
                <mat-error #nameError *ngIf="nameCtrl.errors">
                    {{ getInvalidNameErrorMessage() }} !
                </mat-error>
        </mat-form-field>
        <p></p>
        <mat-form-field appearance="outline">
            <mat-label for="price" class="form-control">Price: </mat-label>
            <mat-hint>The product price  </mat-hint>
            <input matInput placeHolder="Product price" data-testid="price" type="text"
                formControlName="price" type="number" required uppercase>
                <mat-error #priceError *ngIf="priceCtrl.errors">
                    {{ getInvalidPriceErrorMessage() }} !
                </mat-error>
        </mat-form-field>
        <p></p>
        <mat-form-field appearance="outline">
            <mat-label for="currency" class="form-control">Currency: </mat-label>
            <mat-hint>The product currency  </mat-hint>
            <mat-select placeHolder="Product currency" data-testid="currency"
                (selectionChange)="selectionChanged($event)" required>
                <mat-option *ngFor="let currency of mainCurrencies" [value]="currency.value">
                    {{ currency.viewValue }}
                </mat-option>
            </mat-select>
                <mat-error #currencyError *ngIf="currencyCtrl.errors">
                    {{ getInvalidCurrencyErrorMessage() }} !
                </mat-error>
        </mat-form-field>
        <p></p>
        <button mat-raised-button data-testid="submitButton" type="submit"
            color="accent" [disabled]="!productForm.valid">
            <mat-icon>send</mat-icon>
        </button>
    </div>
</form>
```

Figure 5.8 – Implementing the form

This is the detail for the HTML code in the form:

```
<form [formGroup]="productForm" (ngSubmit)="onSubmit()">

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

```
<mat-form-field appearance="outline">

<mat-label for="name" class="form-control">Name: </mat-
label>

<mat-hint>The product name </mat-hint>

<input matInput placeHolder="Product Name" data-
testid="name" type="text"

formControlName="name" required uppercase>

<mat-error #nameError *ngIf="nameCtrl.errors">

{{ getInvalidNameErrorMessage() }} !

</mat-error>

</mat-form-field>

<p></p>

<mat-form-field appearance="outline">

<mat-label for="price" class="form-control">Price: </mat-
label>

<mat-hint>The product price </mat-hint>

<input matInput placeHolder="Product price" data-
testid="price" type="text"

formControlName="price" type="number" required uppercase>

<mat-error #priceError *ngIf="priceCtrl.errors">

{{ getInvalidPriceErrorMessage() }} !

</mat-error>

</mat-form-field>

<p></p>

<mat-form-field appearance="outline">

<mat-label for="currency" class="form-control">Currency:
</mat-label>
```

```
<mat-hint>The product currency </mat-hint>

<mat-select placeHolder="Product currency" data-
testid="currency"

(selectionChange)="selectionChanged($event)" required>

<mat-option *ngFor="let currency of mainCurrencies"
[value]="currency.value">

{{ currency.viewValue }}

</mat-option>

</mat-select>

<mat-error #currencyError *ngIf="currencyCtrl.errors">

{{ getInvalidCurrencyErrorMessage() }} !

</mat-error>

</mat-form-field>

<p></p>

<button mat-raised-button data-testid="submitButton"
type="submit"

color="accent" [disabled]="!productForm.valid">

<mat-icon>send</mat-icon>

</button>

</div>

</form>
```

The form contains an HTML form for submitting Products, using the Reactive-forms API and the html tags provided by the Material library.

It contains three input fields for name, price, and currency, each with a label and validation error message.

The Material library provides advanced html components for input type text (name), input type number (price), and input type select option (currency).

There is also a **Submit** button provided by Material to create the new Product, which is only enabled if all the form fields are valid and disabled if any field is invalid.

Mirroring the backend model

We will use a new class to implement the same model we created in the backend containing the same fields, to facilitate operations in our Angular component.

This model is also written in TypeScript and will be used by the TypeScript code for our component, as we will learn in the next section when we implement the code for the component form.

The new file is called `product.model.ts` and is located in `src/app/shared/model/product.model.ts`.

You can see that it has the same fields as the model in the backend and a constructor method. This is how it looks like, as seen in the figure below:

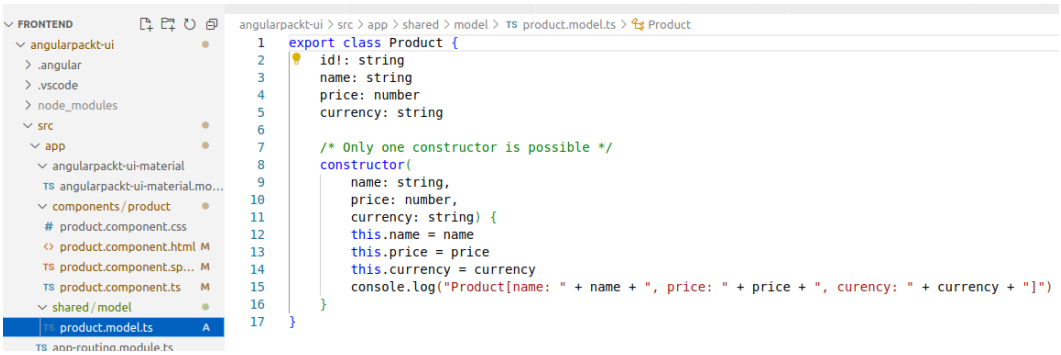


Figure 5.9 – Mirroring the backend model in Angular

This the detail for the code in the `Product` model:

```
export class Product {

id!: string

name: string

price: number

currency: string
```

```
/* Only one constructor is possible */
constructor(
name: string,
price: number,
currency: string) {
this.name = name
this.price = price
this.currency = currency
console.log("Product[name: " + name + ", price: " + price +
", curency: " + currency + "]")
}
}
```

The model declares each field with a type.

It contains a unique constructor method for instantiation with the required fields.

Implementing the component code

Next, we are going to implement the code for the `Product` component in the `product.component.ts` file to provide the dynamic behavior for the form we implemented in the previous section using the reactive-forms API.

This is how it looks like:

product.component.ts

angularpackt-ui > src > app > components > product > product.component.ts > ProductComponent > ngOnInit

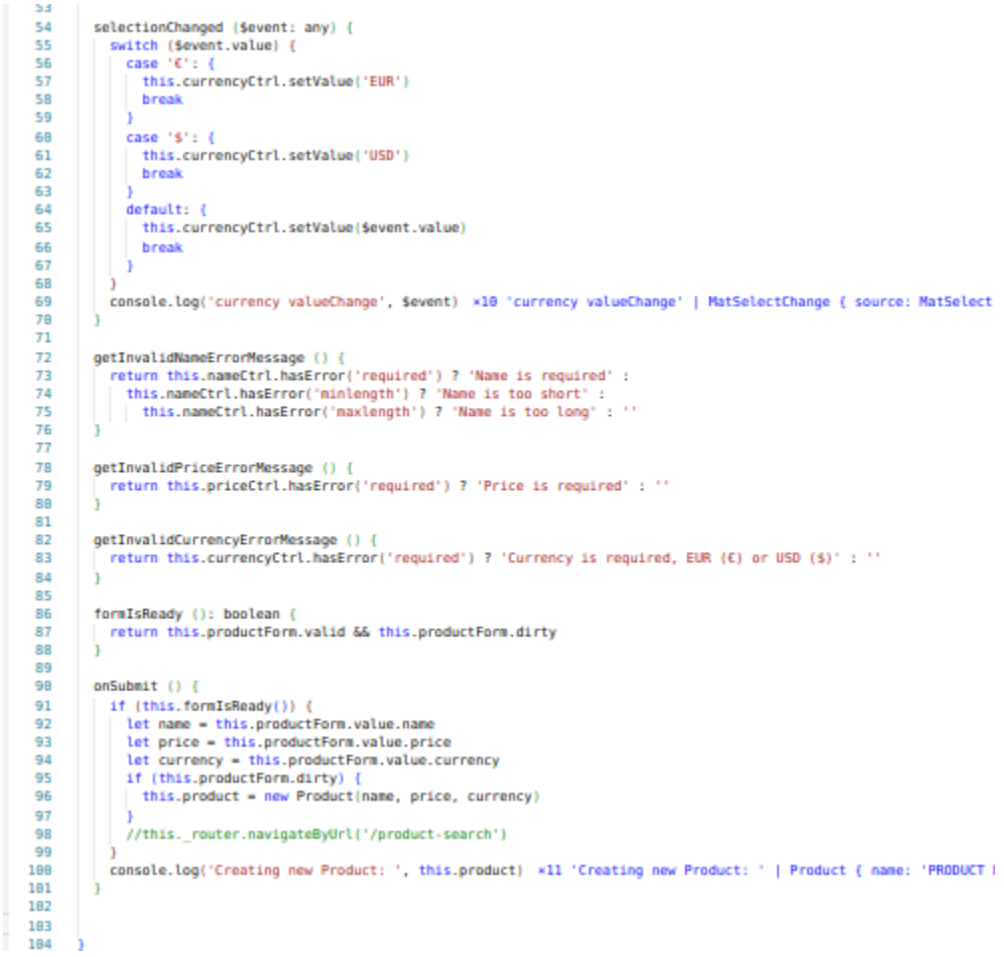
```ts
import { Component, OnInit, ViewChild } from '@angular/core';
import { UntypedFormGroup, UntypedFormBuilder, Validators, UntypedFormControl } from '@angular/forms'
import { Router } from '@angular/router';
import { Currencies, Currency } from 'src/app/shared/convert/currency';
import { Product } from 'src/app/shared/model/product.model';

@Component({
  selector: 'apui-product',
  templateUrl: './product.component.html',
  styleUrls: ['./product.component.css']
})
export class ProductComponent implements OnInit {

  productForm!: UntypedFormGroup;
  nameCtrl!: UntypedFormControl
  priceCtrl!: UntypedFormControl
  currencyCtrl!: UntypedFormControl
  product!: Product;
  mainCurrencies: Currency[] = Currencies.mainCurrencies
  @ViewChild('nameError') nameErrorMessage: any
  @ViewChild('priceError') priceErrorMessage: any
  @ViewChild('currencyError') currencyErrorMessage: any

  constructor(private _router: Router,
    private _formBuilder: UntypedFormBuilder
    ) { }

  ngOnInit(): void {
    this.productForm = this._formBuilder.group({
      name: [{ value: 'Change name here!', disabled: false },
        {
          validators: [Validators.required, Validators.minLength(4), Validators.maxLength(50)],
          updateOn: "blur"
        }
      ],
      price: [{ value: '1', disabled: false },
        {
          validators: [Validators.required],
          updateOn: "blur"
        }
      ],
      currency: [{ value: '€', disabled: false },
        {
          validators: [Validators.required],
          updateOn: "blur"
        }
      ],
    })
    this.nameCtrl = this.productForm.get('name') as UntypedFormControl
    this.priceCtrl = this.productForm.get('price') as UntypedFormControl
    this.currencyCtrl = this.productForm.get('currency') as UntypedFormControl
  }
```



```
  selectionChanged ($event: any) {
    switch ($event.value) {
      case '€': {
        this.currencyCtrl.setValue('EUR')
        break
      }
      case '$': {
        this.currencyCtrl.setValue('USD')
        break
      }
      default: {
        this.currencyCtrl.setValue($event.value)
        break
      }
    }
    console.log('currency valueChange', $event)  ×10 'currency valueChange' | MatSelectChange { source: MatSelect
  }

  getInvalidNameErrorMessage () {
    return this.nameCtrl.hasError('required') ? 'Name is required' :
      this.nameCtrl.hasError('minlength') ? 'Name is too short' :
        this.nameCtrl.hasError('maxlength') ? 'Name is too long' : ''
  }

  getInvalidPriceErrorMessage () {
    return this.priceCtrl.hasError('required') ? 'Price is required' : ''
  }

  getInvalidCurrencyErrorMessage () {
    return this.currencyCtrl.hasError('required') ? 'Currency is required, EUR (€) or USD ($)' : ''
  }

  formIsReady (): boolean {
    return this.productForm.valid && this.productForm.dirty
  }

  onSubmit () {
    if (this.formIsReady()) {
      let name = this.productForm.value.name
      let price = this.productForm.value.price
      let currency = this.productForm.value.currency
      if (this.productForm.dirty) {
        this.product = new Product(name, price, currency)
      }
      //this._router.navigateByUrl('/product-search')
    }
    console.log('Creating new Product: ', this.product)  ×11 'Creating new Product: ' | Product { name: 'PRODUCT
  }


}
```

Figure 5.10 – Implementing the component code

This is the detail for the TypeScript code that powers the form:

```
import { Component, OnInit, ViewChild } from
'@angular/core';

import { UntypedFormGroup, UntypedFormBuilder, Validators,
UntypedFormControl } from '@angular/forms'

import { Router } from '@angular/router';

import { Product } from
'src/app/shared/model/product.model';
```

```
import { Currencies, Currency } from
'src/app/shared/convert/currency';


@Component({
selector: 'apui-product',
templateUrl: './product.component.html',
styleUrls: ['./product.component.css']
})
export class ProductComponent implements OnInit {


productForm!: UntypedFormGroup;
nameCtrl!: UntypedFormControl
priceCtrl!: UntypedFormControl
currencyCtrl!: UntypedFormControl
product!: Product;
mainCurrencies: Currency[]=Currencies.mainCurrencies
@ViewChild('nameError') nameErrorMessage: any
@ViewChild('priceError') priceErrorMessage: any
@ViewChild('currencyError') currencyErrorMessage: any


constructor(private _router: Router,
private _formBuilder: UntypedFormBuilder
) { }


ngOnInit(): void {
```

```
this.productForm = this._formBuilder.group({
name: [{ value: 'Change Name here!', disabled: false },
{
validators: [Validators.required, Validators.minLength(4),
Validators.maxLength(50)],
updateOn: "blur"
}
],
price: [{ value: '1', disabled: false },
{
validators: [Validators.required],
updateOn: "blur"
}
],
currency: [{ value: '€', disabled: false },
{
validators: [Validators.required],
updateOn: "blur"
}
],
})
this.nameCtrl = this.productForm.get('name') as
UntypedFormControl
this.priceCtrl = this.productForm.get('price') as
UntypedFormControl
this.currencyCtrl = this.productForm.get('currency') as
UntypedFormControl
```

```
}
selectionChanged ($event: any) {
switch ($event.value) {
case '€': {
this.currencyCtrl.setValue('EUR')
break
}
case '$': {
this.currencyCtrl.setValue('USD')
break
}
default: {
this.currencyCtrl.setValue($event.value)
break
}
}
console.log('currency valueChange', $event)
}
getInvalidNameErrorMessage () {
return this.nameCtrl.hasError('required') ? 'Name is
required' :
this.nameCtrl.hasError('minlength') ? 'Name is too short' :
this.nameCtrl.hasError('maxlength') ? 'Name is too long' :
''
}
getInvalidPriceErrorMessage () {
```

```
return this.priceCtrl.hasError('required') ?     'Price is
required' : ''

}

getInvalidCurrencyErrorMessage () {

return this.currencyCtrl.hasError('required') ?  'Currency
is required, EUR (€) or USD ($)' : ''

}

formIsReady (): boolean {

return this.productForm.valid && this.productForm.dirty

}

onSubmit () {

if (this.formIsReady()) {

let name = this.productForm.value.name

let price = this.productForm.value.price

let currency = this.productForm.value.currency

if (this.productForm.dirty) {

this.product = new Product(name, price, currency)

}

}

console.log('Creating new Product: ', this.product)

}

}
```

The TypeScript class for the `ProductComponent` implements the `OnInit` lifecycle interface in the `ngOnInit` method.

The TypeScript code in the `ngOninit` method is used for providing the dynamic functionality in the HTML form, and is implemented with a `FormBuilder`, a `FormGroup`, and one `FormControl` for each field.

Each field is declared with a default value and value validators that can be provided by the Forms API, or alternatively we can provide our own custom-made validators.

Because our model requires that each field in the `Product` model is mandatory, these fields have the `Validators.required validator` in their list of provided validators to apply.

The `name` field uses two other validators to restrict the valid length of the field: `Validators.minlength` and `Validators.maxlength` to specify the minimum and maximum allowed number of characters.

The `currency` field uses the `selectionChanged` method to listen to the event when the selection option is changed to assign the value.

The form will be valid for submission only if all the fields in the form are valid. If any field is invalid, the form will automatically be invalid for submission.

The is also one method for each field e.g.: `getInvalidNameErrorMessage()` to provide the different validation error messages for each constraint or validator applied.

The method `formIsReady()` is used, as its name says, to check that the form is valid and ready for submission.

The code is also using the model we created in the previous section for instantiation before submission.

The `onSubmit` method is executed when the submit button in the form is clicked if the form is valid. This method will execute the code to connect to the backend to create the new `Product` record.

Finally, this is how our new component looks like in the browser as shows the figure below:

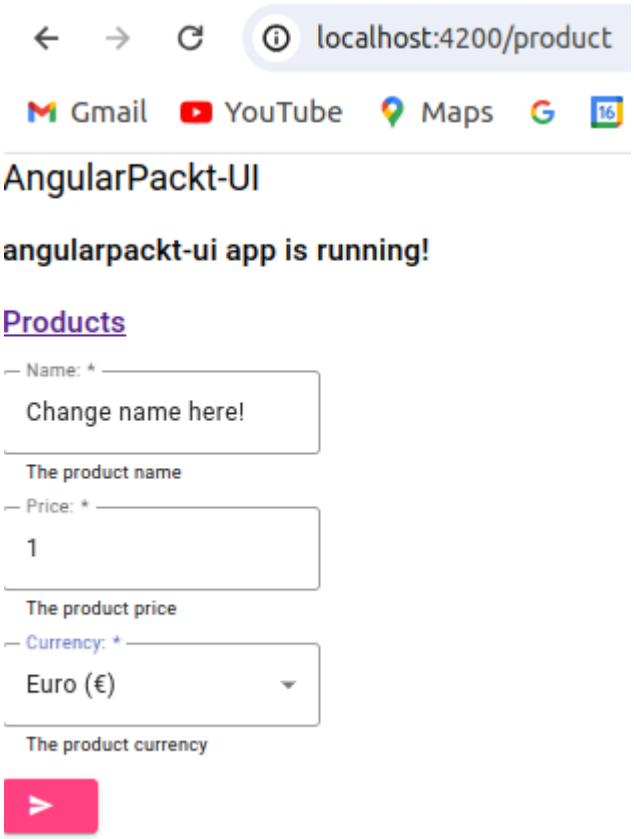


Figure 5.11 – The new component in the browser

Summary

In this chapter, we learned how to install all the necessary tools to build the frontend Angular application. We learned how to install and configure the example provided in this book in our development environment, or alternatively to generate our own Angular application from scratch.

We also learned how to implement a basic Angular component that implements a form using the Angular reactive-forms API and the Angular Material library for enhanced look and feel.

In the next chapter, we will create some Angular services to link our components so that they can connect to the backend application through the backend API we created in the first half of this book.