6 Connecting the Frontend to the Backend REST API with Angular Services

In the previous Chapter of this book, we learned how to implement a frontend SPA with Google's Angular framework and how to build the basic or core part of an Angular application, the Component that manages the form that the users will use to interact with the application. Now we need to implement the code that will interact with the backend to be able to manage the state of the application.

We will concentrate in this chapter on building Angular services, the pattern that will provide the code that the Angular component will use to interact with an external API. In the case of this tutorial, the external API is the DRF API we created in the first part of this book.

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

- **Creating Fake Angular services**
- **Creating Real Angular services**
- **Using Dependency injection for binding components and services**

Creating Angular Services

Whenever you need some code to let any Angular component interact with an external API and manage the state of your application from the frontend (like managing records in the database). Angular provides a useful pattern called 'Service'.

Keep in mind that when implementing the Code for the service you may face two different scenarios:

- **The external API already exists and is accessible all the time.**

  In this case you already know everything about the external API like URLs and parameters. In this scenario you can proceed to implement a real Service that connects to the existing API.

- **The external API is not implemented yet or sometimes is not accessible.**

  In this case you may just have partial or no information at all about the external API. The API may exist but is not always available, and the team can´t use it whenever they want at any time for development. In this scenario you have the alternative that consists of implementing a fake implementation of the service with the information you may have of it, or just assuming how it should look like in the future when the backend team implements it. You can provide this fake implementation to your Angular components that will mimic the behavior of a real Rest API and it will allow you to continue working as if the real API exists.

When creating a new Angular service, the best approach is to use a feature from Typescript that allows to use an abstract class for the service contract (method signatures) and extend the fake and real implementations from it.

Create an abstract class for the service contract

To create the abstract parent class for the Product service first create the following folder structure in `src/app/services/product/`

Inside that folder you can either generate the file manually or with the Angular CLI executing the following command:

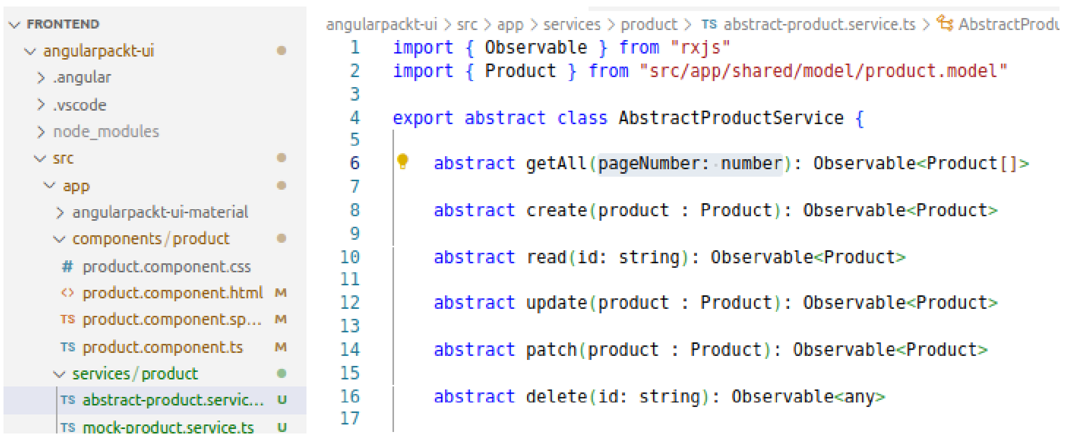
```
$ ng generate class abstract-[model].service
```

For the Product model the command should be like:

```
Full-Stack-Web-Development-with-Django-and-
Angular/frontend/angularpackt-ui/src/app/services/product$
ng generate class abstract-Product.service
```

Alternatively, as we mentioned above, you can also create this abstract class by hand.

Once you have this abstract class for the service, you can type the methods that the external API should implement (the service contract) and leave the implementation empty, as the following image shows:



```
import { Observable } from "rxjs"
import { Product } from "src/app/shared/model/product.model"

export abstract class AbstractProductService {

    abstract getAll(pageNumber: number): Observable<Product[]>

    abstract create(product : Product): Observable<Product>

    abstract read(id: string): Observable<Product>

    abstract update(product : Product): Observable<Product>

    abstract patch(product : Product): Observable<Product>

    abstract delete(id: string): Observable<any>
```

Figure 6.1 - Create Abstract Service class

Here are the details of the source code of the contract for the service:

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

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

```
export abstract class AbstractProductService {


     abstract getAll(pageNumber: number):
Observable<Product[]>


     abstract create(product : Product):
     Observable<Product>


     abstract read(id: string): Observable<Product>


     abstract update(product : Product):
     Observable<Product>


     abstract patch(product : Product): Observable<Product>


     abstract delete(id: string): Observable<any>


}
```

The contract of the service should implement API endpoints for creating, reading, updating, partial updating, deleting and finding all Product records in the database.

As you may have noticed most methods return an Observable of the Model.

Observable

Observables are a technique for event handling, asynchronous programming, and handling multiple values emitted over time. You define a function for publishing values, called the source, but that function is not executed until the consumer subscribes to the observable by calling the observable's 'subscribe' method.

Angular uses the RxJs Library for observables.
Variables that are defined as type Observable usually follow the naming convention of adding the '$' character at the end of the variable name: E.g.: products$: Observable<Product[]>

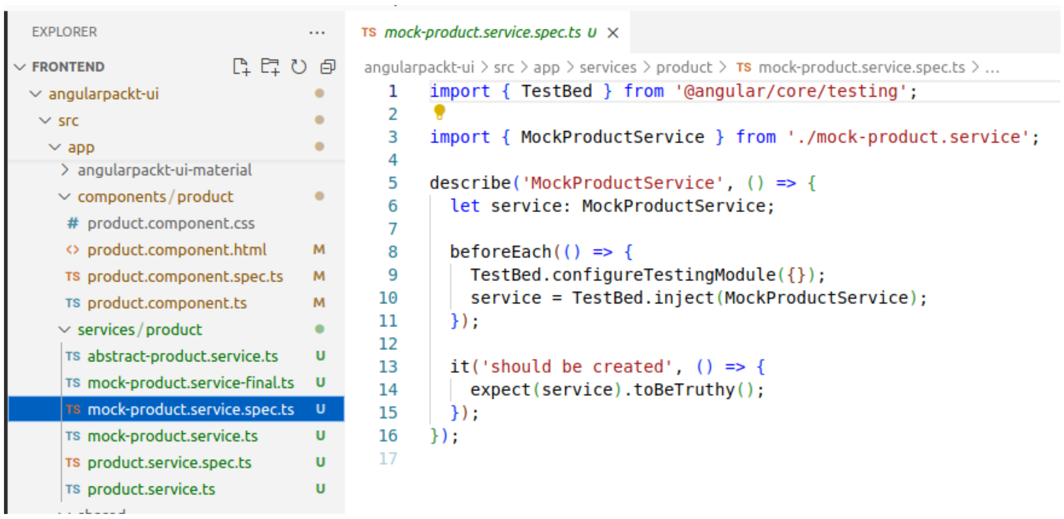
Create an implementation for the fake service that matches the contract

Now you can create the new files for the fake implementation of the service contract defined before in the abstract class.

To create the initial code for the fake implementation of the service you can generate it with the angular CLI in a similar way as we did for the abstract class `AbstractProductService`

You can generate the initial source code for the mock implementation of the service typing the following command with the angular CLI:

```
Full-Stack-Web-Development-with-Django-and-Angular/frontend/angularpackt-ui/src/app/services/product$
ng generate service mock-Product

CREATE src/app/services/product/mock-product.service.spec.ts (383 bytes)

CREATE src/app/services/product/mock-product.service.ts (140 bytes)
```


EXPLORER
FRONTEND
angularpackt-ui
src
app
angularpackt-ui-material
components / product
product.component.css
product.component.html M
product.component.spec.ts M
product.component.ts M
services / product
abstract-product.service.ts U
mock-product.service-final.ts U
mock-product.service.spec.ts U
mock-product.service.ts U
product.service.spec.ts U
product.service.ts U
mock-product.service.spec.ts U
angularpackt-ui > src > app > services > product > mock-product.service.spec.ts > ...

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

import { MockProductService } from './mock-product.service';

describe('MockProductService', () => {
  let service: MockProductService;

  beforeEach(() => {
    TestBed.configureTestingModule({});
    service = TestBed.inject(MockProductService);
  });

  it('should be created', () => {
    expect(service).toBeTruthy();
  });
});
```

Figure 6.2 - Generate mock service class with Angular CLI

This is how the initial generation of the code for the mock service looks like.

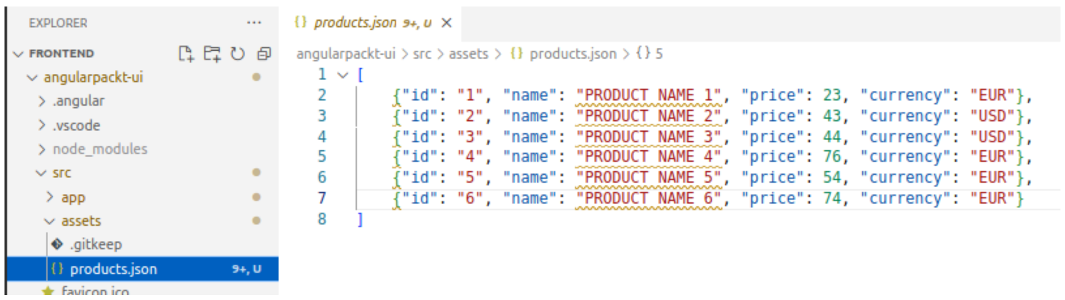
Later we will implement the methods that match the contract specified in the abstract class for the service we created before.

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


@Injectable({
providedIn: 'root'
})
export class MockProductService {
      constructor() { }
}
```

Now let's implement the different methods for the mock. A common way to do it is by loading the data in **Json** format from a file instead of calling the real external API.

We need to manually create a new file containing the data source for the records that will replace the records stored in a real database.

Create a new file under the `src/assets/` folder and call it: `products.json`



```
[
    {"id": "1", "name": "PRODUCT NAME 1", "price": 23, "currency": "EUR"},
    {"id": "2", "name": "PRODUCT NAME 2", "price": 43, "currency": "USD"},
    {"id": "3", "name": "PRODUCT NAME 3", "price": 44, "currency": "USD"},
    {"id": "4", "name": "PRODUCT NAME 4", "price": 76, "currency": "EUR"},
    {"id": "5", "name": "PRODUCT NAME 5", "price": 54, "currency": "EUR"},
    {"id": "6", "name": "PRODUCT NAME 6", "price": 74, "currency": "EUR"}
]
```

Figure 6.3 - Fake datasource file

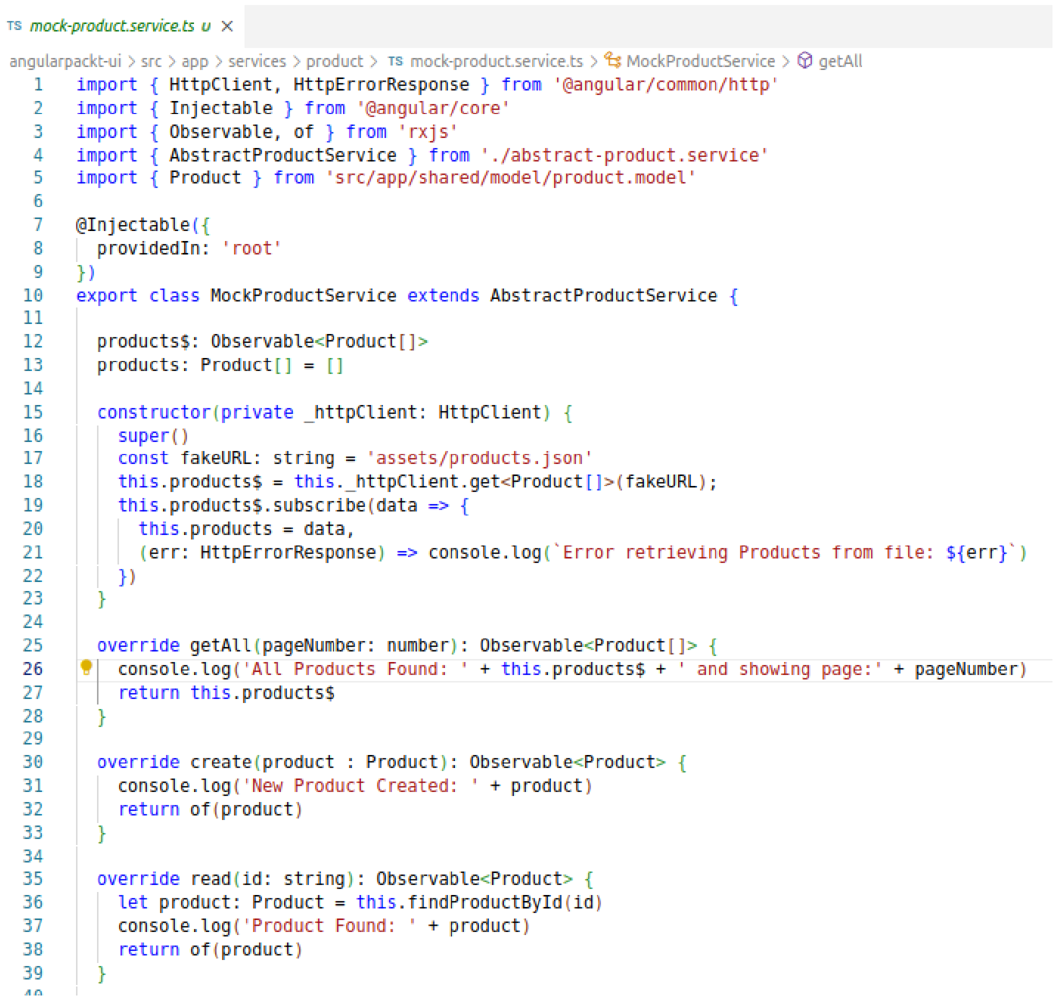
This is how the code for the list of Fake records in Json format looks like:

```
[
        {"id": "1", "name": "Product Name 1", "price": 23,
        "currency": "EUR"},

        {"id": "2", "name": "Product Name 2", "price": 34,
        "currency": "USD"},

        ...
]
```

This represents an array of Products in json format that mimics the response in Json format for a call to an external API to retrieve all the Product records stored in a real database.

And once we have a **datasource** of records to feed our fake service, we can implement the methods that will use this data as if they were obtained from a real external API.

We will implement the fake service by overriding the same methods that declared in the abstract class for the service by extending the `AbstractProductService` class.
See figure below:



TS mock-product.service.ts U ×
angularpackt-ui > src > app > services > product > TS mock-product.service.ts > MockProductService > getAll

```
import { HttpClient, HttpErrorResponse } from '@angular/common/http'
import { Injectable } from '@angular/core'
import { Observable, of } from 'rxjs'
import { AbstractProductService } from './abstract-product.service'
import { Product } from 'src/app/shared/model/product.model'

@Injectable({
  providedIn: 'root'
})
export class MockProductService extends AbstractProductService {

  products$: Observable<Product[]>
  products: Product[] = []

  constructor(private _httpClient: HttpClient) {
    super()
    const fakeURL: string = 'assets/products.json'
    this.products$ = this._httpClient.get<Product[]>(fakeURL);
    this.products$.subscribe(data => {
      this.products = data,
      (err: HttpErrorResponse) => console.log(`Error retrieving Products from file: ${err}`)
    })
  }

  override getAll(pageNumber: number): Observable<Product[]> {
    console.log('All Products Found: ' + this.products$ + ' and showing page:' + pageNumber)
    return this.products$
  }

  override create(product : Product): Observable<Product> {
    console.log('New Product Created: ' + product)
    return of(product)
  }

  override read(id: string): Observable<Product> {
    let product: Product = this.findProductById(id)
    console.log('Product Found: ' + product)
    return of(product)
  }
```

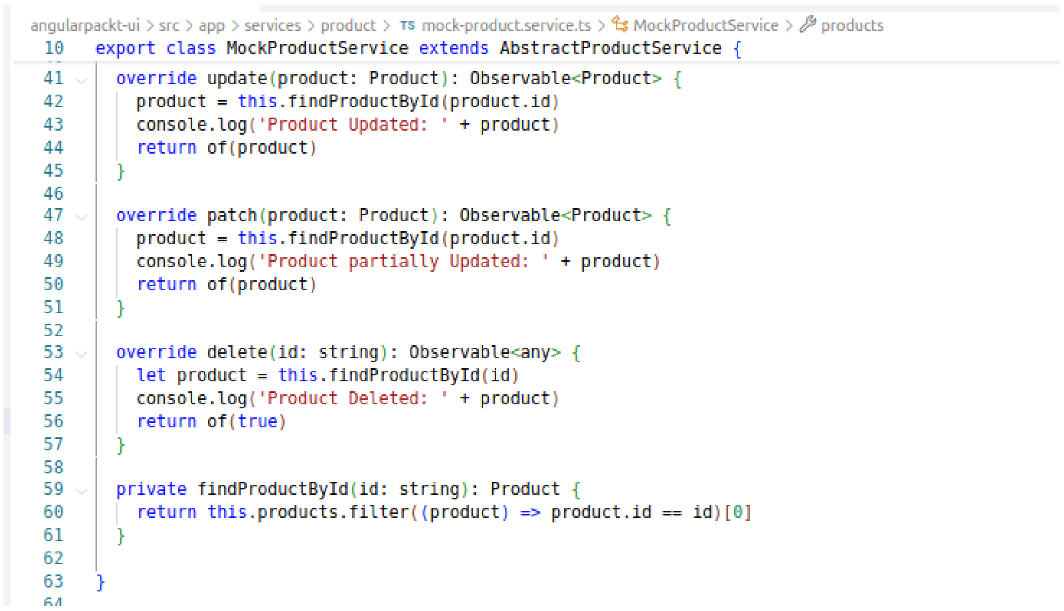

```
angularpackt-ui > src > app > services > product > TS mock-product.service.ts > MockProductService > products
export class MockProductService extends AbstractProductService {
  override update(product: Product): Observable<Product> {
    product = this.findProductById(product.id)
    console.log('Product Updated: ' + product)
    return of(product)
  }

  override patch(product: Product): Observable<Product> {
    product = this.findProductById(product.id)
    console.log('Product partially Updated: ' + product)
    return of(product)
  }

  override delete(id: string): Observable<any> {
    let product = this.findProductById(id)
    console.log('Product Deleted: ' + product)
    return of(true)
  }

  private findProductById(id: string): Product {
    return this.products.filter((product) => product.id == id)[0]
  }

}
```

Figure 6.4 - Implementing the mock service

This is how the source code for the mock service looks like:

```
import { HttpClient, HttpErrorResponse } from
'@angular/common/http'

import { Injectable } from '@angular/core'

import { Observable, of } from 'rxjs'

import { AbstractProductService } from './abstract-
product.service'

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


@Injectable({

providedIn: 'root'
```

```
})

export class MockProductService extends
AbstractProductService {

    products$: Observable<Product[]>

    products: Product[] = []

    constructor(private _httpClient: HttpClient) {

        super();

        const url: string = 'assets/products.json'

        this.products$ =
        this._httpClient.get<Product[]>(url);

        this.products$.subscribe(data => {

        this.products = data,

        (err: HttpErrorResponse) => console.log(`Error
        retrieving Products from file: ${err}`)

        })

    }

    override getAll(): Observable<Product[]> {

        console.log('All Products Found: ' +
        this.products$)

        return this.products$

    }

    override create(product : Product):
    Observable<Product> {

        console.log('New Product Created: ' + product)
        return of(product)
    }

    override read(id: string): Observable<Product> {
        let product: Product = this.findProductById(id)
        console.log('Product Found: ' + product)
        return of(product)
    }

    override update(product: Product): Observable<Product>
    {
        product = this.findProductById(product.id)
        console.log('Product Updated: ' + product)
        return of(product)
    }

    override patch(product: Product): Observable<Product>
    {
        product = this.findProductById(product.id)
        console.log('Product partially Updated: ' +
        product)
        return of(product)
    }

    override delete(id: string): Observable<any> {
        let product = this.findProductById(id)
        console.log('Product Deleted: ' + product)

        return of(true)

    }


    private findProductById(id: string): Product {

        return this.products.filter((product) =>
        product.id == id)[0]

    }


}
```

The mock service extends the abstract service: `AbstractProductService` class, therefore must override all the methods in that contract class, keeping the same method signature.

The service will be able to be used by other Angular components thanks to a mechanism called Dependency injection, using the `@Injectable` annotation.

Dependency Injection

Dependency Injection or DI is a common pattern used by many frameworks to decouple components from each other by avoiding instantiation of classes. The abstract classes or interfaces are injected instead of the concrete implementations, therefore making the code independent of the actual implementations.
This pattern is the key for easy maintenance in applications as it allows to produce code that is simpler to maintain and easier to test.

Thanks to DI it is easy to change the implementation of a Service used by other components. We will see how it works later in this chapter when we switch from the mock service to the real service used by the `ProductComponent`.

The mock service is loading product records from a file in Json format in the code for the constructor with the Angular `Httpclient` tool that returns an **Observable** as the value for the `products$` variable.

The code contains fake implementations for the methods declared in the `AbstractProductService` class, that is: `getAll`, `create`, `read`, `update`, `patch` and `delete`. There is also one extra helper method called `findProductById` that is just used internally in the code for the implementation of these methods but is not part of the contract for the service.

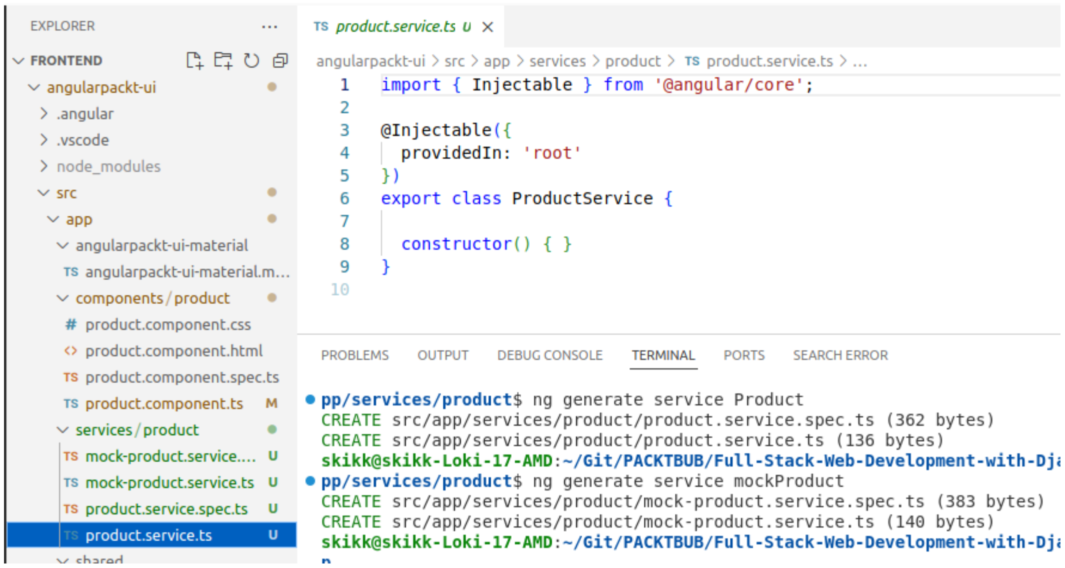
Create an implementation for the real service that matches the contract

Now you can create the new files for the real implementation of the service contract defined before in the abstract class.

To create the initial code for the real implementation of the service you can generate it with the angular CLI in a similar way as we did for the abstract class `AbstractProductService`

You can generate the initial source code for the real implementation of the service by typing the following command with angular CLI:

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

CREATE src/app/services/product/product.service.spec.ts
(362 bytes)

CREATE src/app/services /product/product.service.ts (136
bytes)
```



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

@Injectable({
  providedIn: 'root'
})
export class ProductService {

  constructor() { }
}
```

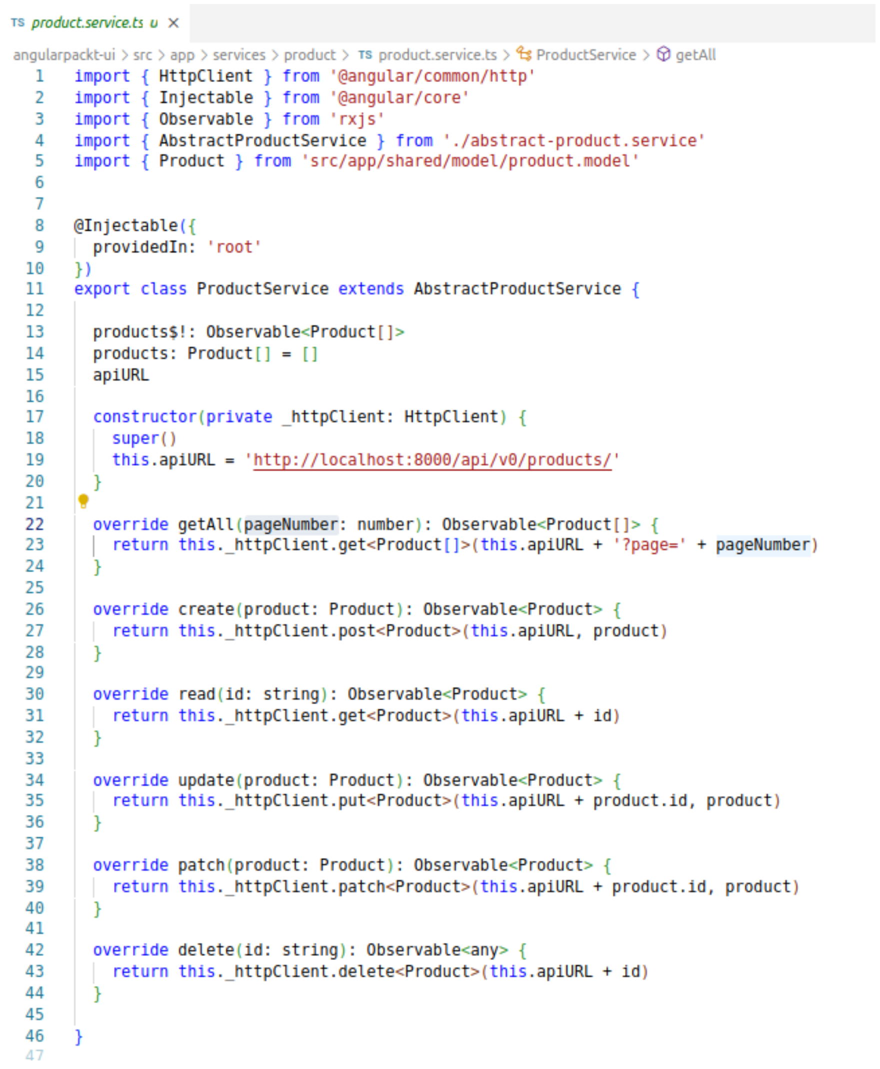
```
● pp/services/product$ ng generate service Product
CREATE src/app/services/product/product.service.spec.ts (362 bytes)
CREATE src/app/services/product/product.service.ts (136 bytes)
skikk@skikk-Loki-17-AMD:~/Git/PACKTBUB/Full-Stack-Web-Development-with-Dja
● pp/services/product$ ng generate service mockProduct
CREATE src/app/services/product/mock-product.service.spec.ts (383 bytes)
CREATE src/app/services/product/mock-product.service.ts (140 bytes)
skikk@skikk-Loki-17-AMD:~/Git/PACKTBUB/Full-Stack-Web-Development-with-Dja
```

Figure 6.4 - Generate real service class with Angular CLI

This is how the initial generation of the code for the real service looks like. Later we will implement the methods that match the contract specified in the abstract class for the service we created before.

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

@Injectable({
     providedIn: 'root'
})
export class ProductService {
     constructor() { }
}
```

We will implement the real service by overriding the same methods that declared in the abstract class for the service by extending the `AbstractProductService` class. See figure below:


product.service.ts U
angularpackt-ui > src > app > services > product > product.service.ts > ProductService > getAll

```
import { HttpClient } from '@angular/common/http'
import { Injectable } from '@angular/core'
import { Observable } from 'rxjs'
import { AbstractProductService } from './abstract-product.service'
import { Product } from 'src/app/shared/model/product.model'


@Injectable({
  providedIn: 'root'
})
export class ProductService extends AbstractProductService {

  products$!: Observable<Product[]>
  products: Product[] = []
  apiURL

  constructor(private _httpClient: HttpClient) {
    super()
    this.apiURL = 'http://localhost:8000/api/v0/products/'
  }

  override getAll(pageNumber: number): Observable<Product[]> {
    return this._httpClient.get<Product[]>(this.apiURL + '?page=' + pageNumber)
  }

  override create(product: Product): Observable<Product> {
    return this._httpClient.post<Product>(this.apiURL, product)
  }

  override read(id: string): Observable<Product> {
    return this._httpClient.get<Product>(this.apiURL + id)
  }

  override update(product: Product): Observable<Product> {
    return this._httpClient.put<Product>(this.apiURL + product.id, product)
  }

  override patch(product: Product): Observable<Product> {
    return this._httpClient.patch<Product>(this.apiURL + product.id, product)
  }

  override delete(id: string): Observable<any> {
    return this._httpClient.delete<Product>(this.apiURL + id)
  }

}
```

Figure 6.5 - Implementing the real service

This is how the source code for the real service looks like:

```
import { HttpClient } from '@angular/common/http'

import { Injectable } from '@angular/core'
import { Observable } from 'rxjs'

import { AbstractProductService } from './abstract-
product.service'

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



@Injectable({

providedIn: 'root'

})

export class ProductService extends AbstractProductService
{

    products$!: Observable<Product[]>

    products: Product[] = []

    apiURL


    constructor(private _httpClient: HttpClient) {

        super()

        this.apiURL =
        'http://localhost:8000/api/v0/products/'

    }


    override getAll(pageNumber: number):
    Observable<Product[]> {

        return
        this._httpClient.get<Product[]>(this.apiURL +
        '?page=' + pageNumber)
    }

    override create(product: Product): Observable<Product>
    {
        return
        this._httpClient.post<Product>(this.apiURL,
        product)
    }

    override read(id: string): Observable<Product> {
        return this._httpClient.get<Product>(this.apiURL
    + id)
    }

    override update(product: Product): Observable<Product>
    {
        return this._httpClient.put<Product>(this.apiURL
        + product.id, product)
    }


    override patch(product: Product): Observable<Product>
    {
        return
        this._httpClient.patch<Product>(this.apiURL +
        product.id, product)
    }


    override delete(id: string): Observable<any> {
        return
        this._httpClient.delete<Product>(this.apiURL +
        id)
    }
}
```

The real service extends the abstract service: `AbstractProductService` class, therefore must override all the methods in that contract class, keeping the same method signature.

The service will be able to be used by other Angular components thanks to the Dependency injection (DI) mechanism that we described before for the fake implementation using the `@Injectable` annotation.

Thanks to DI it is easy to change the implementation of a service used by other components. We will later see how it works in this chapter when we switch from the mock service to the real service used by the `ProductComponent`.

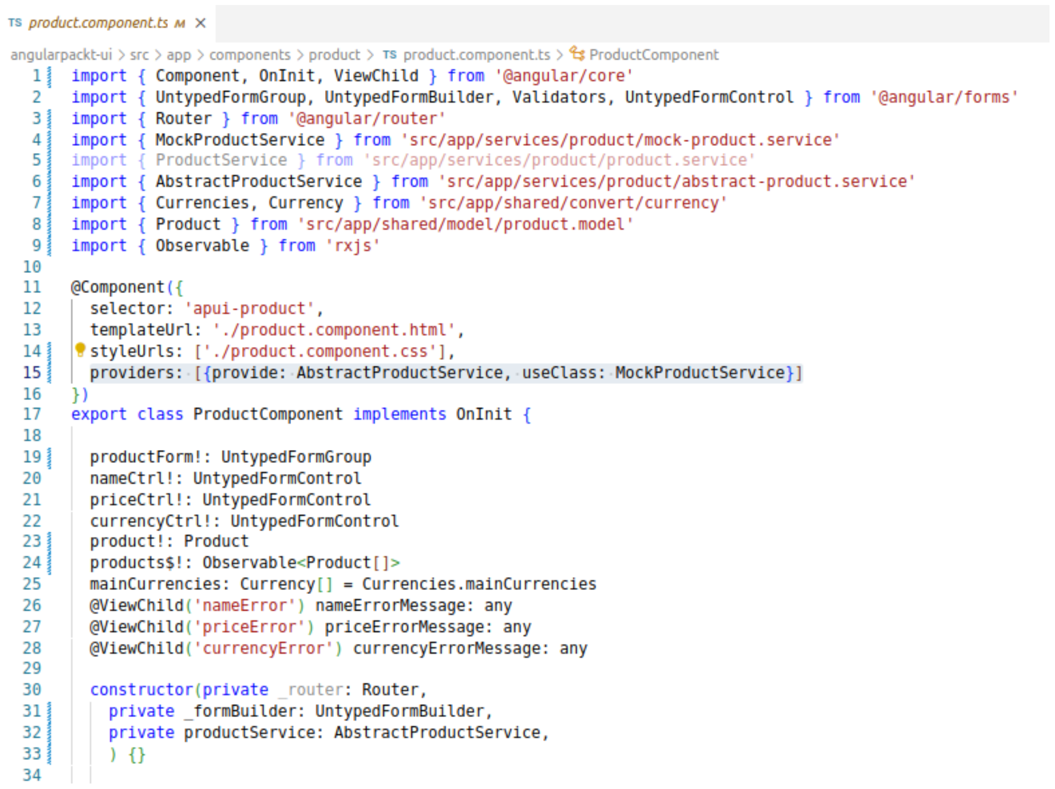
The real service is connecting to the real Rest API provided by the Backend that we created in the first part of this book with the Angular `Httpclient` tool that returns an **Observable** as the value for the `products$` variable.

The code contains real implementations for the methods declared in the `AbstractProductService` class, that is: `getAll`, `create`, `read`, `update`, `patch` and `delete`.

Inject an implementation for the service in the Component

Next, we will bind the Angular Component with the angular Service. For this, we will use Dependency injection to connect the component to one of the service implementations created in the previous sections, the fake or the real implementation of the service.

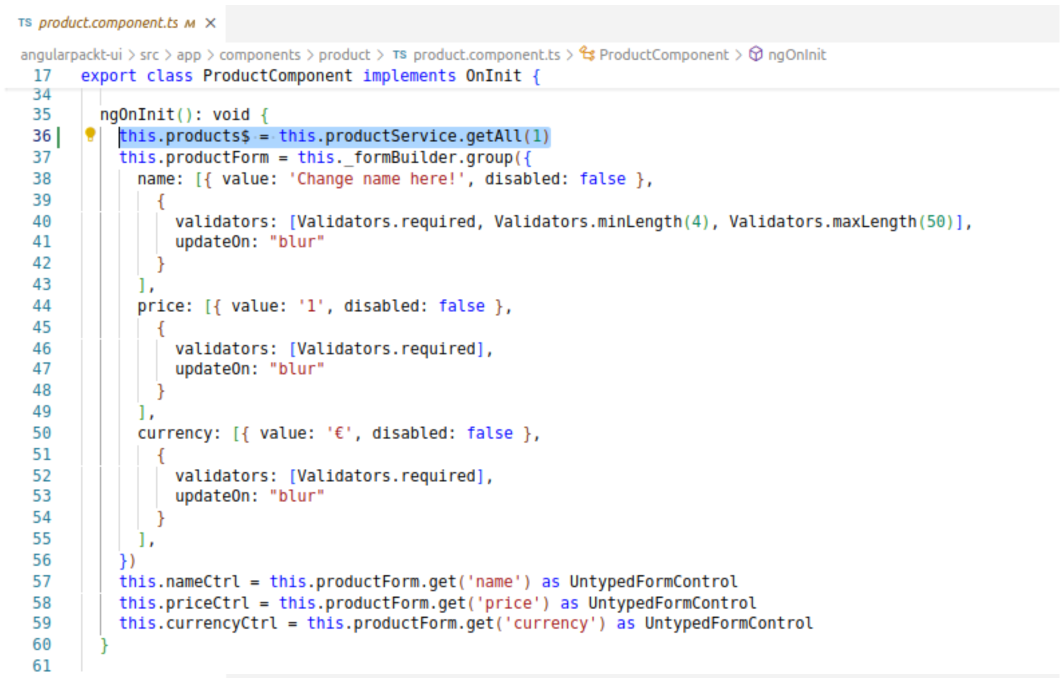
See figure below:



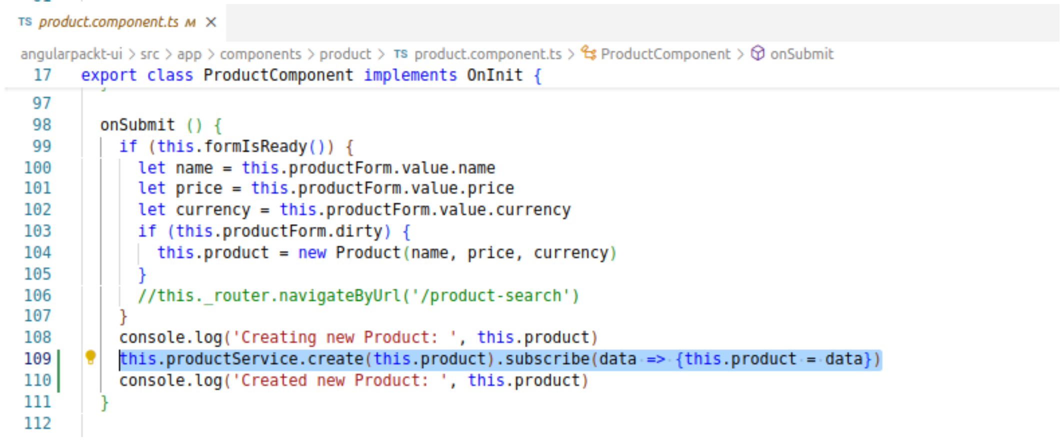
TS product.component.ts M ×

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

```
import { Component, OnInit, ViewChild } from '@angular/core'
import { UntypedFormGroup, UntypedFormBuilder, Validators, UntypedFormControl } from '@angular/forms'
import { Router } from '@angular/router'
import { MockProductService } from 'src/app/services/product/mock-product.service'
import { ProductService } from 'src/app/services/product/product.service'
import { AbstractProductService } from 'src/app/services/product/abstract-product.service'
import { Currencies, Currency } from 'src/app/shared/convert/currency'
import { Product } from 'src/app/shared/model/product.model'
import { Observable } from 'rxjs'

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

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

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

```


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

```
export class ProductComponent implements OnInit {

  ngOnInit(): void {
    this.products$ = this.productService.getAll(1)
    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
  }

```


TS product.component.ts M ×
angularpackt-ui > src > app > components > product > TS product.component.ts > ProductComponent > onSubmit

```
export class ProductComponent implements OnInit {

  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)
    this.productService.create(this.product).subscribe(data => {this.product = data})
    console.log('Created new Product: ', this.product)
  }

```

Figure 6.6 - Injecting the Service in the Component

This is how the source code for the component, using the fake service looks like (only the relevant part of the code is shown):

- Both the MockProductService and the real ProductService classes are imported in the component:

```
import { MockProductService } from
'src/app/services/product/mock-product.service'

import { ProductService } from
'src/app/services/product/product.service'
```

- The Service is injected in the component using one implementation (MockProductService):

```
@Component({
    selector: 'apui-product',
    templateUrl: './product.component.html',
    styleUrls: ['./product.component.css'],
    providers: [{provide: AbstractProductService, useClass:
    MockProductService}]
})
```

- A paginated list of products is loaded on the `onInit()` lifecycle method for the Component, to show the list of all products on the top of the form:

```
ngOnInit(): void {

    this.products$ = this.productService.getAll(1)

}
```

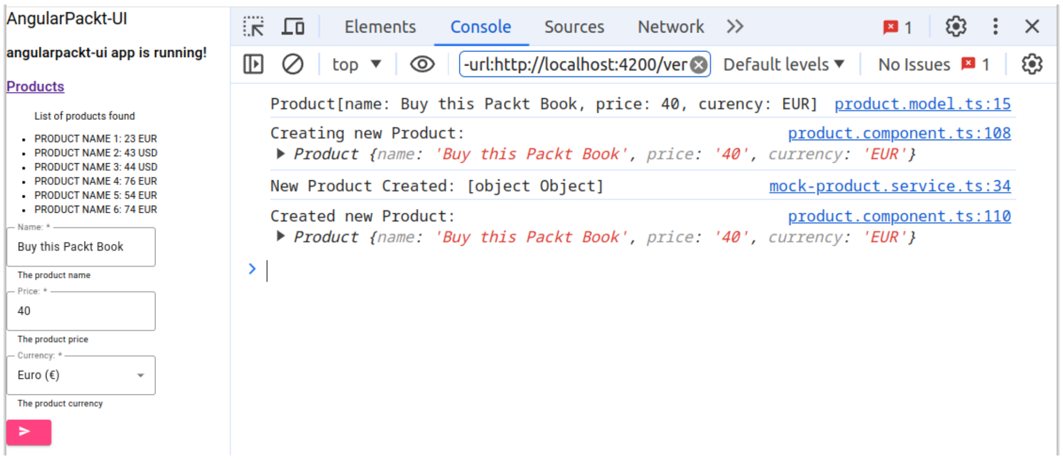
- The method to create a new Product is called on the service in the `onSubmit()` method from the form:

```
onSubmit () {

    this.productService.create(this.product).subscribe(dat
    a => {this.product = data})

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

}
```

This is how the form looks like, now showing the list of all Product records plus the form to create a new Product.

You can also see the log produced by the angular console when a new Product is submitted for creation.

See figure below:


AngularPackt-UI

angularpackt-ui app is running!

Products

List of products found

- PRODUCT NAME 1: 23 EUR
- PRODUCT NAME 2: 43 USD
- PRODUCT NAME 3: 44 USD
- PRODUCT NAME 4: 76 EUR
- PRODUCT NAME 5: 54 EUR
- PRODUCT NAME 6: 74 EUR

Name: * Buy this Packt Book — The product name

Price: * 40 — The product price

Currency: * Euro (€) — The product currency

```
Product[name: Buy this Packt Book, price: 40, curency: EUR]  product.model.ts:15
Creating new Product:                                        product.component.ts:108
▶ Product {name: 'Buy this Packt Book', price: '40', currency: 'EUR'}
New Product Created: [object Object]                         mock-product.service.ts:34
Created new Product:                                         product.component.ts:110
▶ Product {name: 'Buy this Packt Book', price: '40', currency: 'EUR'}
```

Figure 6.7 - Angular Service interaction from the UI

Switch to another implementation of the service in the Component

Once you decide that you no longer need the fake implementation for the Service because the real implementation is ready and stable to be consumed, or just to occasionally test the integration with the real Rest API, you can easily switch to the real implementation of the service thanks to dependency injection with a single line of code.

See figure below:

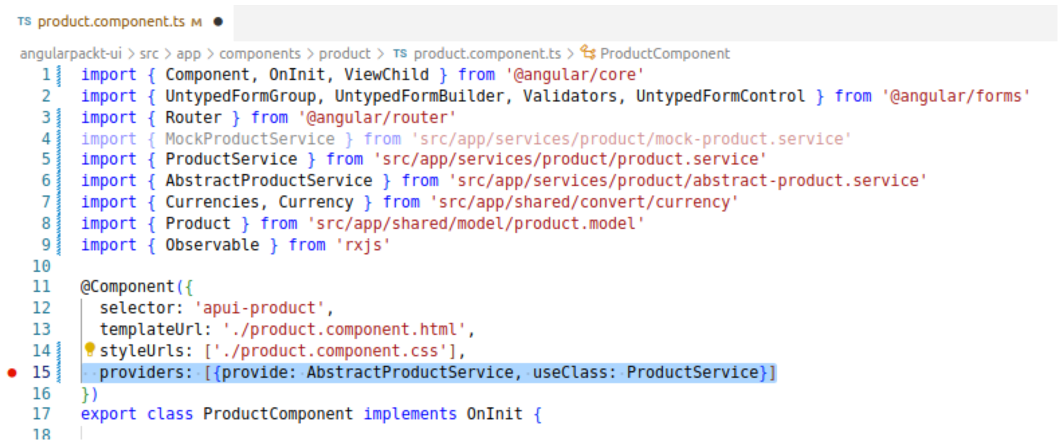

```
product.component.ts M ●
angularpackt-ui > src > app > components > product > product.component.ts > ProductComponent
import { Component, OnInit, ViewChild } from '@angular/core'
import { UntypedFormGroup, UntypedFormBuilder, Validators, UntypedFormControl } from '@angular/forms'
import { Router } from '@angular/router'
import { MockProductService } from 'src/app/services/product/mock-product.service'
import { ProductService } from 'src/app/services/product/product.service'
import { AbstractProductService } from 'src/app/services/product/abstract-product.service'
import { Currencies, Currency } from 'src/app/shared/convert/currency'
import { Product } from 'src/app/shared/model/product.model'
import { Observable } from 'rxjs'

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

Figure 6.8 - Switching the service implementation with DI

This is the one-liner change to switch to the real service implementation:

```
@Component({
        selector: 'apui-product',
        templateUrl: './product.component.html',
        styleUrls: ['./product.component.css'],
        providers: [{provide: AbstractProductService, useClass: ProductService}]
})
```

As the code shows it is as simple as changing the mock class for the real class.

Summary

In this chapter we learned how to implement an important pattern used in Angular called 'Service' used to communicate the frontend with an external Rest API.

We also saw that we can implement a fake service while the real service is not ready yet.

Finally, we learned how to easily switch from one service implementation to another with the help of Dependency Injection.

In the next chapter, we will learn to restrict access to different Angular components based on the User information stored on the Angular session with an approach called Role based access control **RBAC**.