Angular training

Service, HTTP and RxJS

URL Git for this project

https://github.com/wanmuz86/ecommerce-angular-nov.git

Services in Angular

We have been bundling all our business logic into a single component.

Our application might become unmaintainable very soon.

Angular Dependency Injection (DI) is a mechanism to declare and consume our dependencies across the application with minimum effort and optimal results

We will see how to create an Angular application that is correctly structured to enforce the Separation of Concerns (SoC) pattern using services.

https://angular.io/guide/architecture-services

Service in Angular

- Introducing the Angular DI
- Creating our first Angular service
- Providing dependencies across the application

Injecting services in the component tree

UI and Route

components/header

components/footer

pages/products : products/:id

pages/products-list : ''

pages/add-product : /create-product

Introduction to Angular DI

DI is an application design pattern we also come across in other languages, such as C# and Java.

As our applications grow and evolve, each code entity will internally require instances of other objects, better known as **dependencies**. Passing such dependencies to the consumer code entityis known as an **injection**, and it also entails the participation of another code entity, called the **injector**.

The injector is responsible for instantiating and bootstrapping the required dependencies to be ready for use when injected into a consumer. The consumer knows nothing about howto instantiate its dependencies and is only aware of the interface they implement to use them.

Previous products website

In real-world applications, we rarely work with static data. It usually comes from a back end API or some other service.

- The list of products is tightly coupled with the component. Angular components are responsible for the presentation logic and should not be concerned with how to get data.

They only need to display it in the HTML template.

Thus, they should delegate business logic to services to handle such tasks.

MVCSP - MVC (Model , View Controller)

View (HTML, CSS)

User Interface of the website

Controller (.TS)

The logic of a website.

Service / Provider

- Connection to data source
- Connection to API
- Connection to Local Database (local storage)
- Firebase

Data source
For eg: API

Product Service

products : Product[]

- addProduct
- getAllProducts
- getProductById

Product List

- Show all Array of products

Product by Id

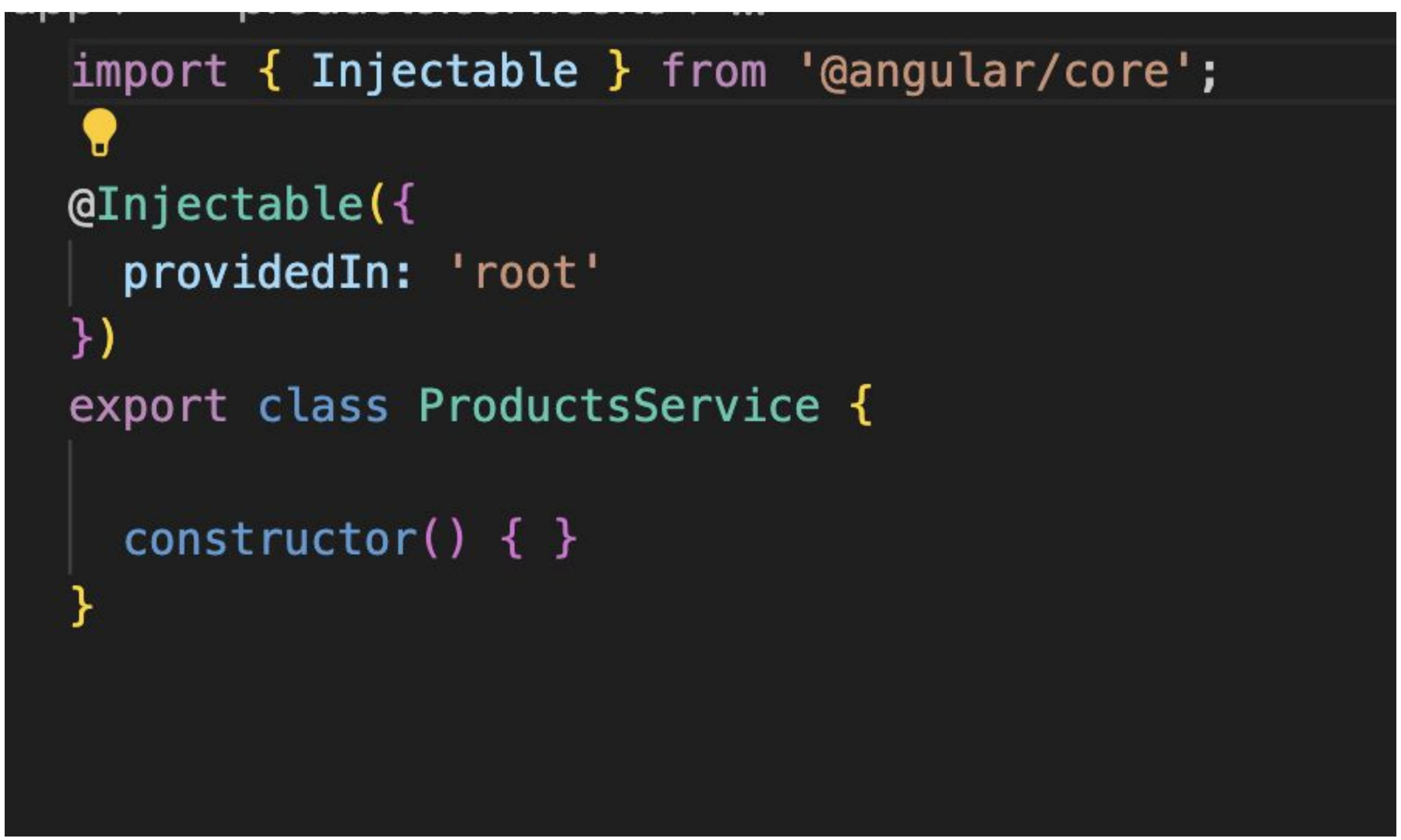
- SHow more info of the chosen product

Add Product

- Add an item in the product list

Creating an Angular Service

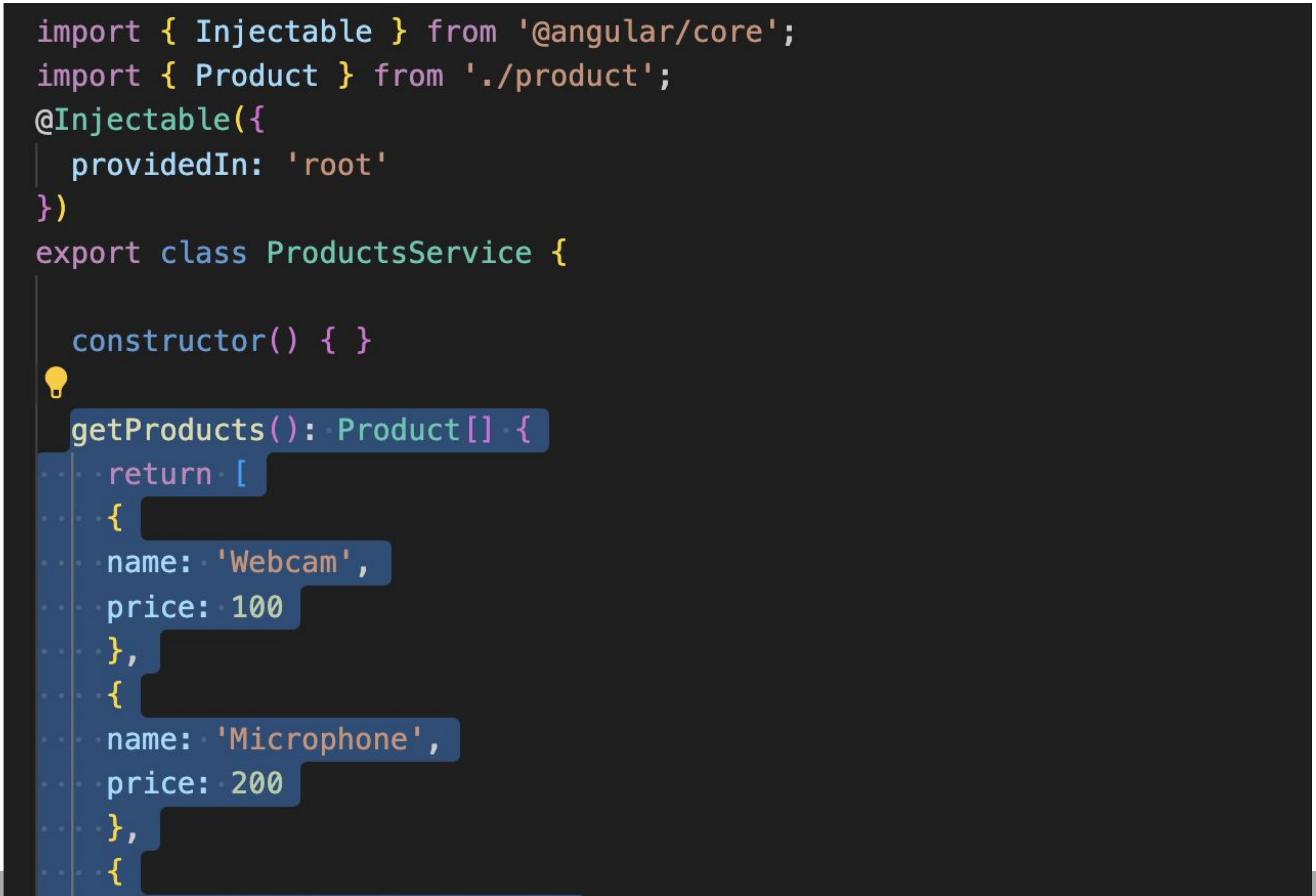
```
ng generate service product

ng generate service services/product
```



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

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

  constructor() { }
}
```

Creating an Angular Service (2)

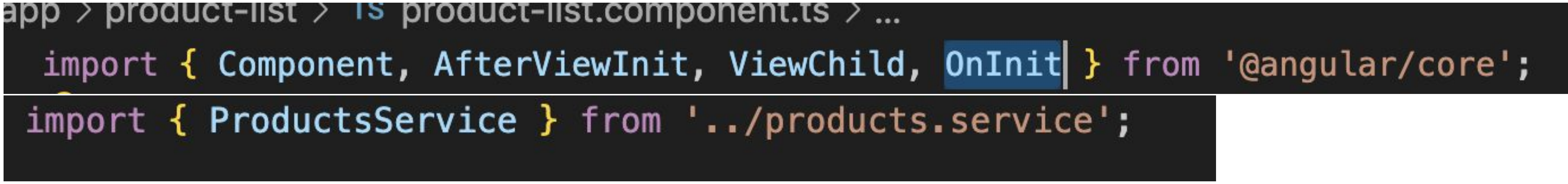
product.service.ts



```
import { Injectable } from '@angular/core';
import { Product } from './product';
@Injectable({
  providedIn: 'root'
})
export class ProductsService {

  constructor() { }

  getProducts(): Product[] {
    return [
    {
    name: 'Webcam',
    price: 100
    },
    {
    name: 'Microphone',
    price: 200
    },
    {
```

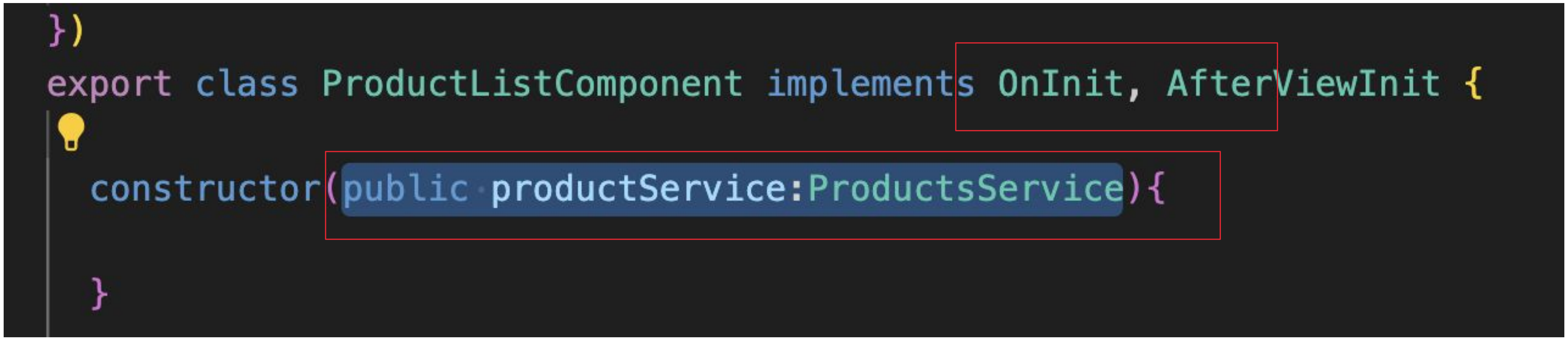
Creating an Angular Service (3)

product-list.component.ts



app > product-list > TS product-list.component.ts > ...

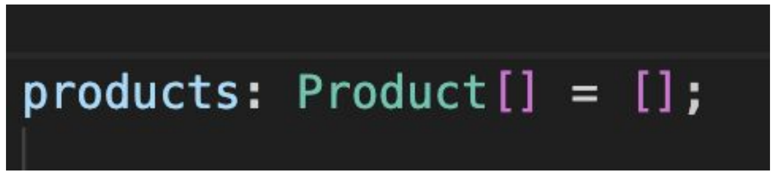
```
 import { Component, AfterViewInit, ViewChild, OnInit } from '@angular/core';
import { ProductsService } from '../products.service';
```



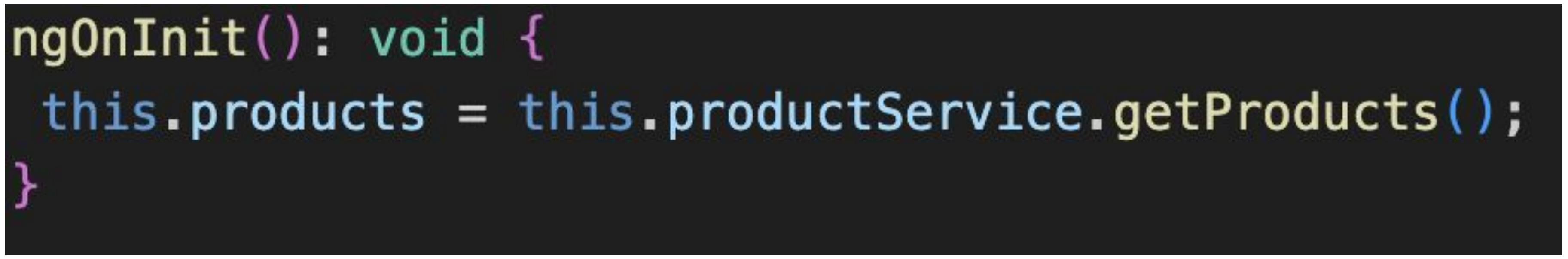
```
})
export class ProductListComponent implements OnInit, AfterViewInit {

  constructor(public productService:ProductsService){

  }
```

Creating an Angular Service (4)

product-list.component.ts



```
products: Product[] = [];
```



```
ngOnInit(): void {
 this.products = this.productService.getProducts();
}
```

Observables and RxJS

RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using Observables, making it easier to compose asynchronous or callback-based code.

https://rxjs.dev/

https://playcode.io/rxjs

https://codecraft.tv/courses/angular/reactive-programming-with-rxjs/observables-and-rxjs/

Key concept of RxJS

- Observable: Represents a stream of data over time. It can emit multiple values asynchronously.
- Observer: Listens to the Observables. It reacts to the data stream by defining functions like `next`, `error`, and `complete`.
- Operators: Functions used to manipulate Observables. They allow transforming, filtering, combining, and handling errors in data streams.
- Subscription: Represents the execution of an Observable. It can be used to unsubscribe from the Observable to prevent memory leaks.

Creating observables

- Creation Functions: RxJS provides various creation functions like `of`, `from`, `interval`, `fromEvent`, etc., to create Observables from different data sources.
- Custom Observables: You can create custom Observables using the `Observable` constructor or by using the `new Observable()` syntax.

Subscribing to observables

- Use the `subscribe()` method to start listening to an Observable.
- Inside `subscribe()`, define functions to handle emitted values, errors, and completion of the Observable.

Observables operator

- Transformation Operators: Examples include `map`, `pluck`, `switchMap`, `mergeMap`, etc., which transform the values emitted by an Observable.
- Filtering Operators: Examples include `filter`, `take`, `takeUntil`, `distinctUntilChanged`, etc., which filter the values emitted by an Observable.
- Combination Operators: Examples include `merge`, `concat`, `combineLatest`, `forkJoin`, etc., which combine multiple Observables into a single Observable.

Error handling

- Use operators like `catchError`, `retry`, `finalize`, etc., to handle errors in Observables.
- Errors can be propagated downstream to subscribers, where they can be handled accordingly.

RxJS use case

- HTTP Requests: RxJS is commonly used to handle asynchronous HTTP requests in Angular applications.
- Event Handling: RxJS simplifies event handling in web applications by providing operators like `fromEvent`.
- Real-time Data Processing: RxJS is suitable for real-time data processing tasks where data streams need to be processed asynchronously.
- Reactive Form in Angular

Example RxJS : creation and subscri

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

// CREATION OF OBSERVABLE
const observable = interval(1000); // Every 1s it will bring out number 1,2,3,4,5

// CREATION OF OBSERVER/SUBSCRIVER
const subscription = observable.subscribe(
 data => console.log(data),
 error => console.log(error),
 () => console.log('Complete')
)
// IF we subscribe to an observable, we need to unsubscribe / ngOnDestroy
// There is an async filter that can be used in HTML to manage observable object
where you can call it right away and unsubscribe automatically

setTimeout(()=>{
subscription.unsubscribe();
```

Example : pipe and map

```
import {Observable, of} from 'rxjs';
import { map } from 'rxjs/operators';

// To transform a data (string,number, array) into observable of() -> A
stream of Array
const numbersObservable = of([1,2,3,4,5]);

// pipe - Is used when we want to run an operation from an observable
// map [inside] -> similar to yesterday's concept, it will take a
value, each of the value, it will ** 2
// map() - > To transform it back into an observable
const transformedObservable = numbersObservable.pipe(map(numbers =>
numbers.map(num=>num**2)))

// we subscribed to transformedObservable, for each of the response
```

Example : of and tap

```
import {of} from 'rxjs';
import { tap } from 'rxjs/operators';

// Creating an observable of 1,2,3,4,5
// nomenclature best practice , observable you will see $ at the end of the name
const sources$ = of(1,2,3,4,5);

// Tap is an operation that you can do multiple thing
// pipe here means I am going to do an observable operation on the data stream
// You can log whatever value that you receive from the data source
// You can transform it at the same time
const tapped$ = sources$.pipe(
 tap(value => console.log('Original value:',value)), // Passed value
 tap(value=> console.log('Squared value: ', value*value)) // Logged value
);

const subscribed = tapped$.subscribe(
 value=> console.log('Received value', value),
 error => console.log("error")
```

Introducing the Angular HTTP client

The built-in HTTP client of the Angular framework is a separate Angular library that resides in the @angular/common npm package under the http namespace.

The Angular CLI installs this package by default when creating a new Angular project.

The HttpClientModule class provides various Angular services we can use to handle asynchronous HTTP communication.

The most basic is the HttpClient service, which provides a robust API and abstracts all operations required to handle asynchronous connections through various HTTP methods.

Its implementation was considered carefully to ensure that developers feel at ease while developing solutions that take advantage of this class.

https://angular.io/guide/understanding-communicating-with-http

Introducing the Angular HTTP client

The HttpClient service have access to various methods to perform common HTTP request operations, such as GET, POST, PUT, and every existing HTTP verb. •

•get: This performs a GET operation to fetch data.

•post: This performs a POST operation to add new data.

•put/patch: This performs a PUT/PATCH operation to update existing data.

delete: This performs a DELETE operation to remove existing data.

URL Used later

Get all products: https://fakestoreapi.com/products

Get product of particular id: https://fakestoreapi.com/products/1

Add a product : https://fakestoreapi.com/products [POST]

Edit a product: https://fakestoreapi.com/products/1 [PATCH/PUT]

Delete a product https://fakestoreapi.com/products/1 [DELETE]

Authentication

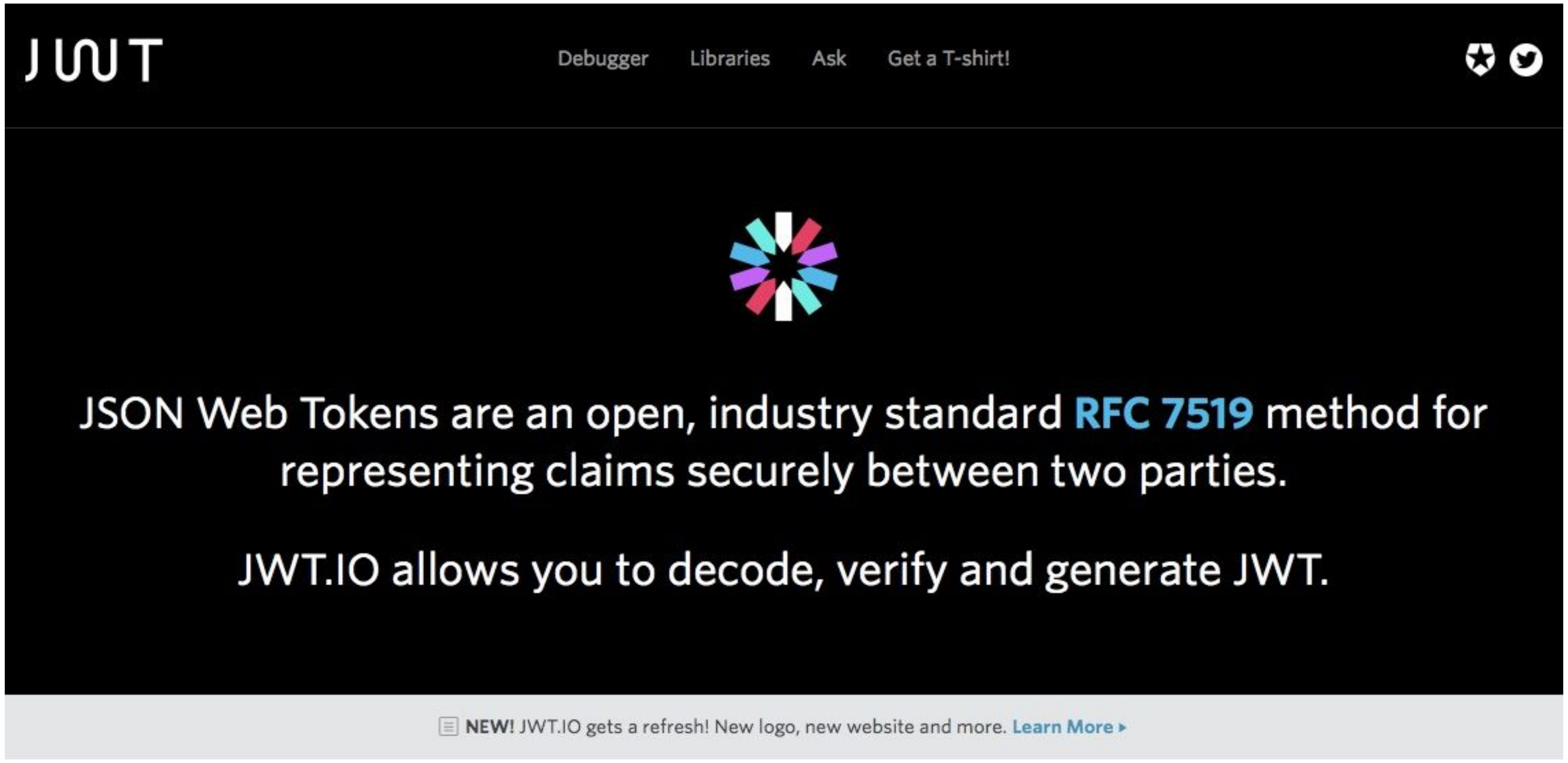
https://fakestoreapi.com/auth/login [POST]

Username and password

username: "mor_2314",

password: "83r5^_"

[hardcode this]

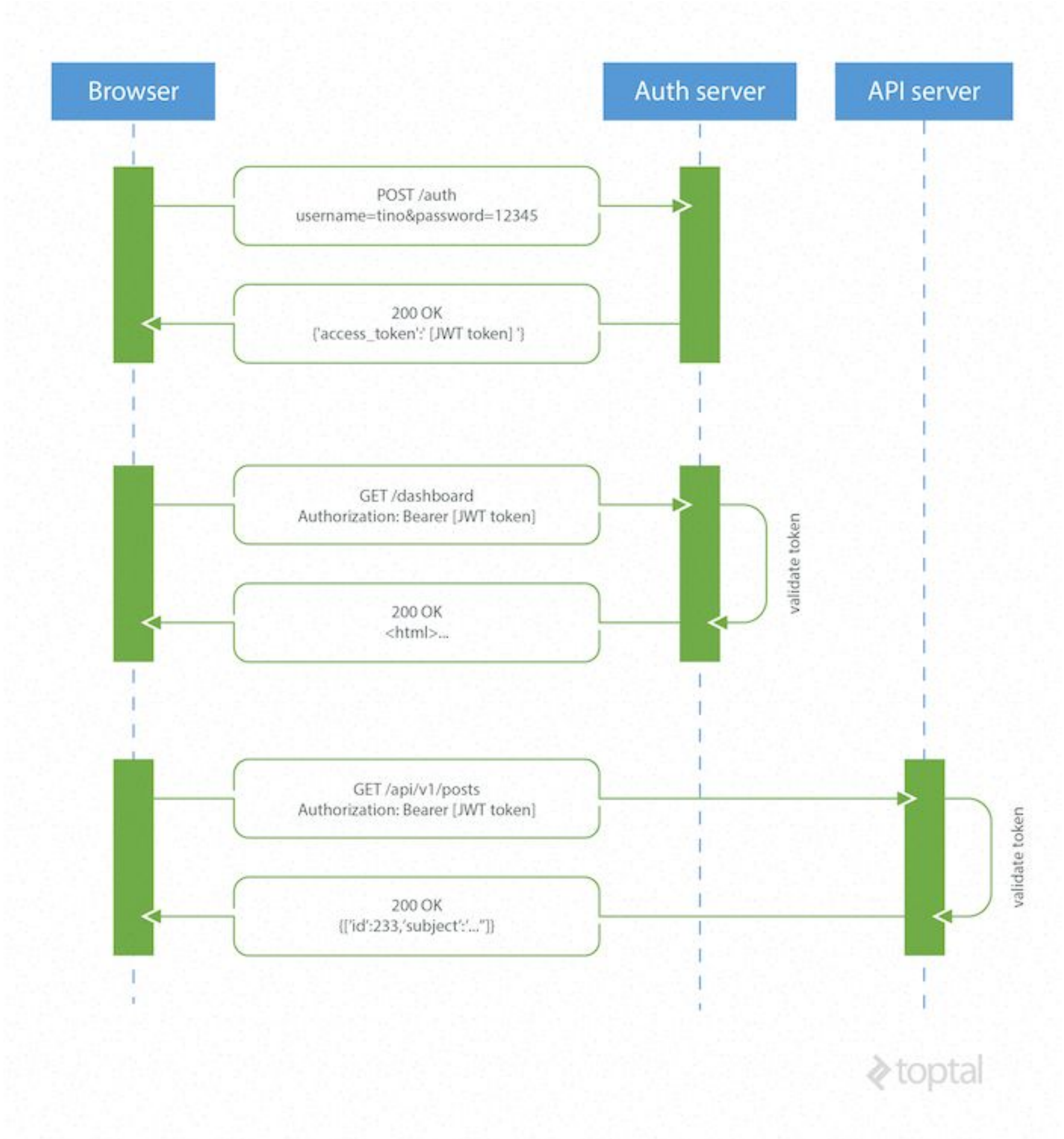


JSON Web Tokens are an open, industry standard RFC 7519 method for representing claims securely between two parties.

JWT.IO allows you to decode, verify and generate JWT.

NEW! JWT.IO gets a refresh! New logo, new website and more. Learn More ▸

Authentication with JWT



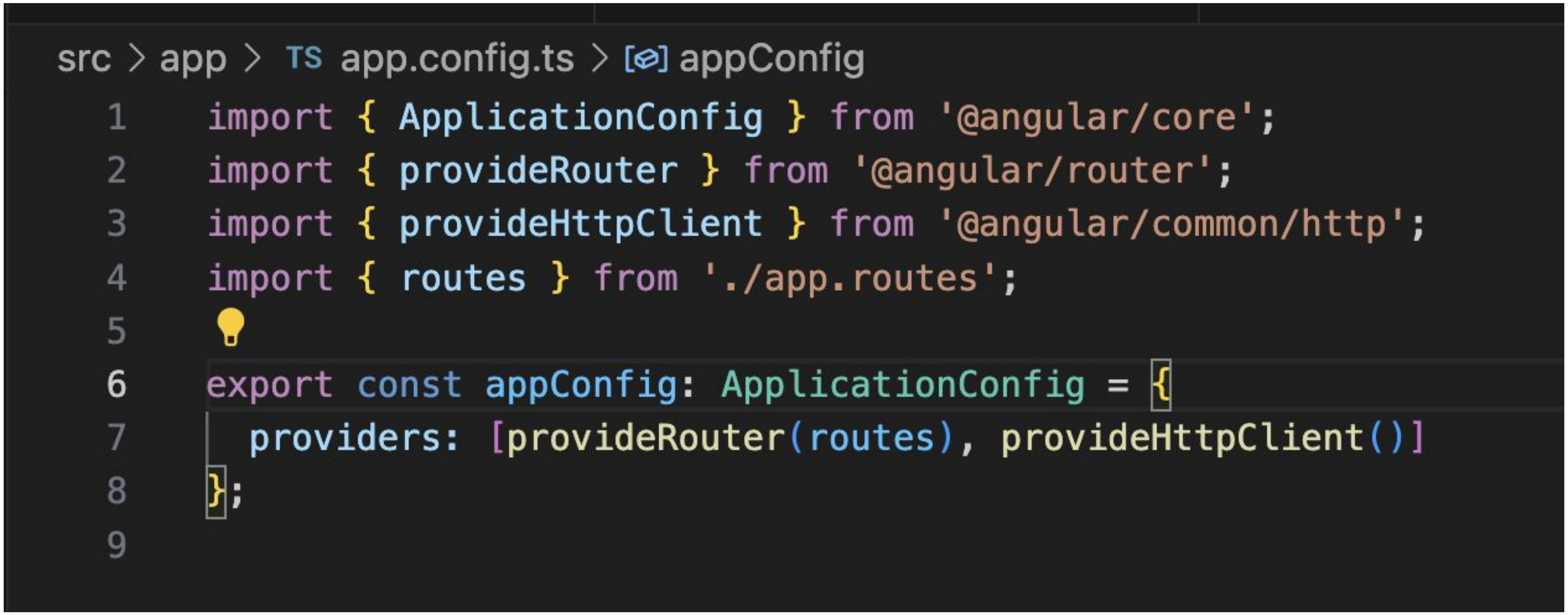
Browser
Auth server
API server
POST /auth
username=tino&password=12345
200 OK
{'access_token':' [JWT token] '}
GET /dashboard
Authorization: Bearer [JWT token]
validate token
200 OK
<html>...
GET /api/v1/posts
Authorization: Bearer [JWT token]
validate token
200 OK
[{'id':233,'subject':'...'}]
toptal

Authentication flow

1. Client makes a request once by sending their login credentials and password;
2. Server validates the credentials and, if everything is right, it returns to the client a JSON with a token that encodes data from a user logged into the system;
3. Client, after receiving this token, can store it the way it wants, whether via LocalStorage, Cookie or other client-side storage mechanisms;
4. Every time the client accesses a route that requires authentication, it will only send this token to the API to authenticate and release consumption data;
5. Server always validate this token to allow or block a customer request.

Fetching data through HTTP

app.config.ts

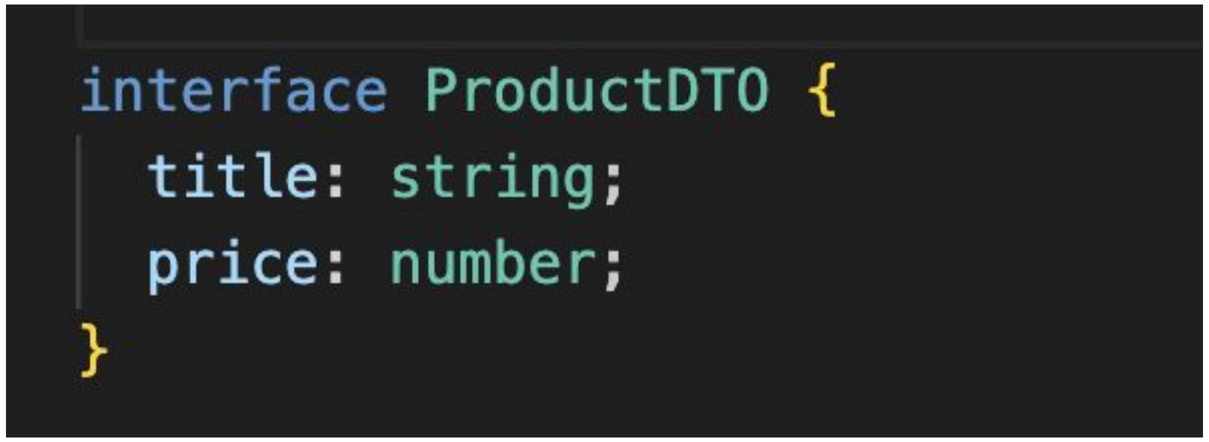


src > app > TS app.config.ts > appConfig

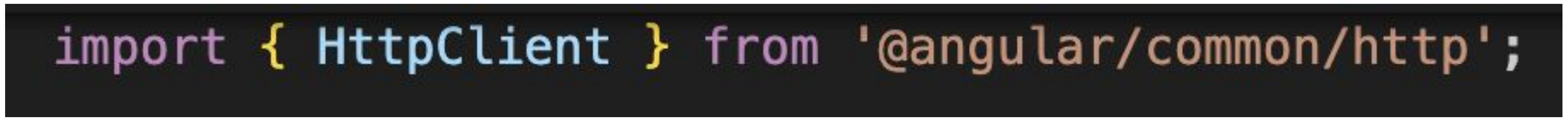
```typescript
import { ApplicationConfig } from '@angular/core';
import { provideRouter } from '@angular/router';
import { provideHttpClient } from '@angular/common/http';
import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [provideRouter(routes), provideHttpClient()]
};
```

Fetching data through HTTP (1)

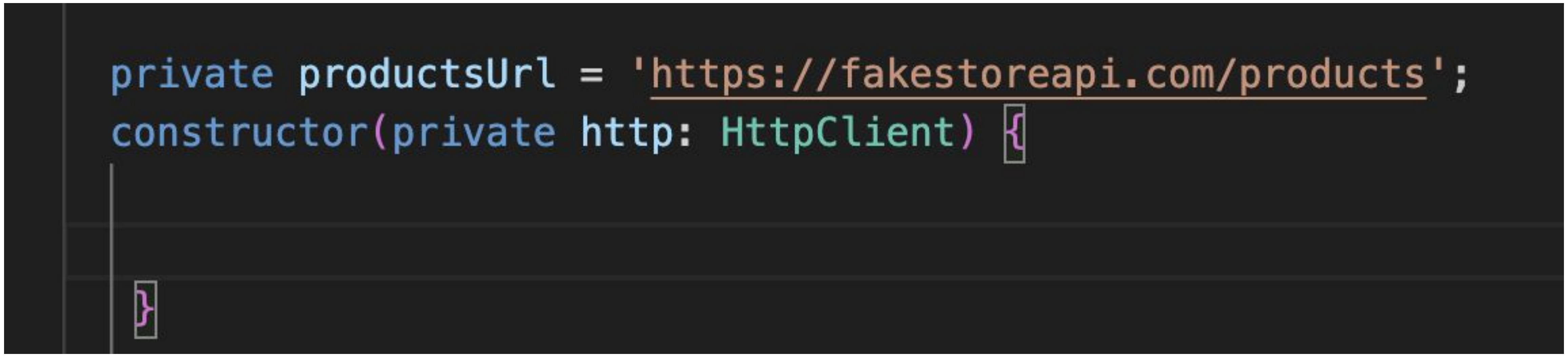
product.service.ts



```
interface ProductDTO {
  title: string;
  price: number;
}
```



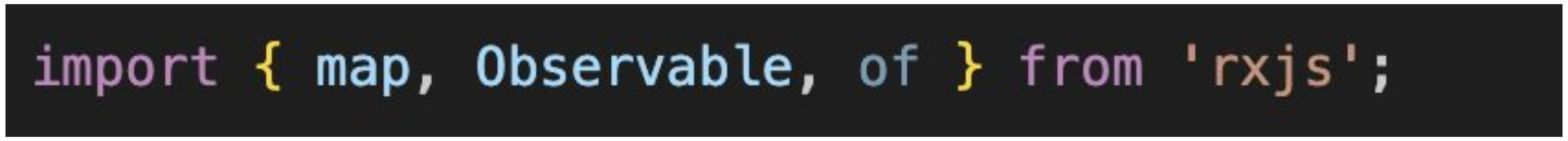
```
import { HttpClient } from '@angular/common/http';
```



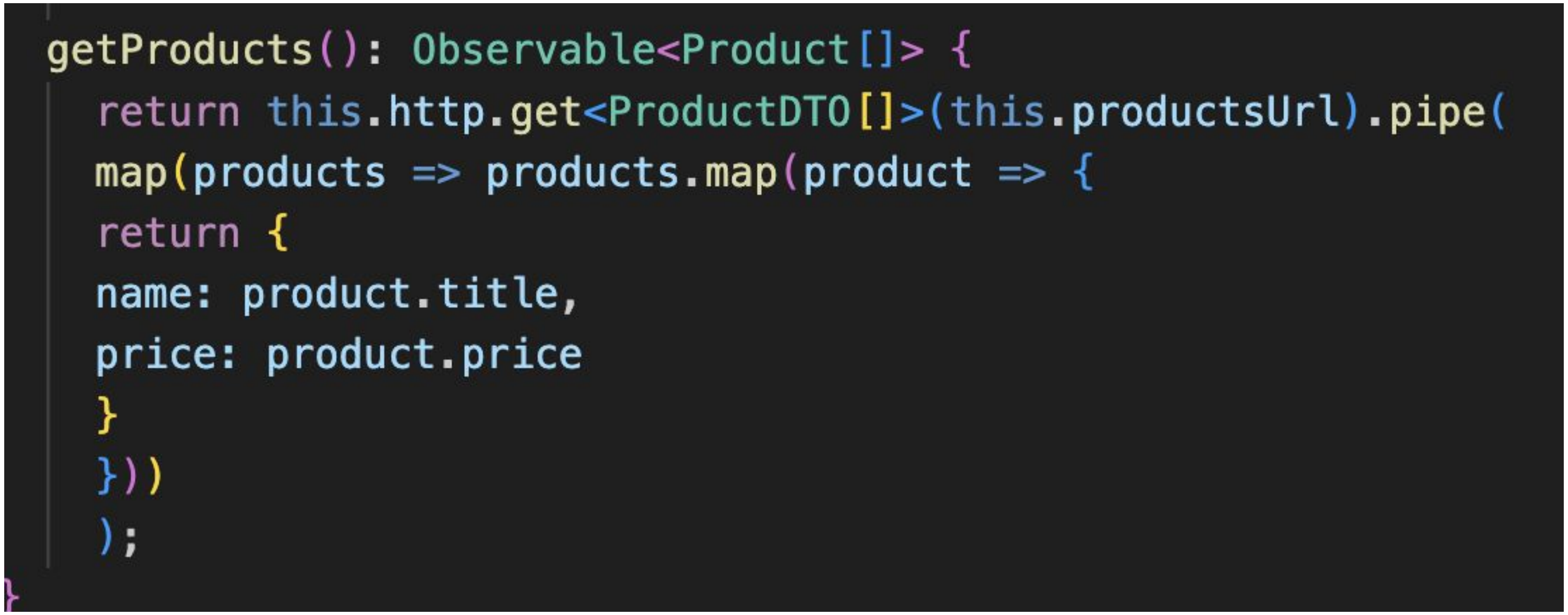
```
  private productsUrl = 'https://fakestoreapi.com/products';
  constructor(private http: HttpClient) {


  }
```

Fetching data through HTTP (2)

product.service.ts



```
import { map, Observable, of } from 'rxjs';
```



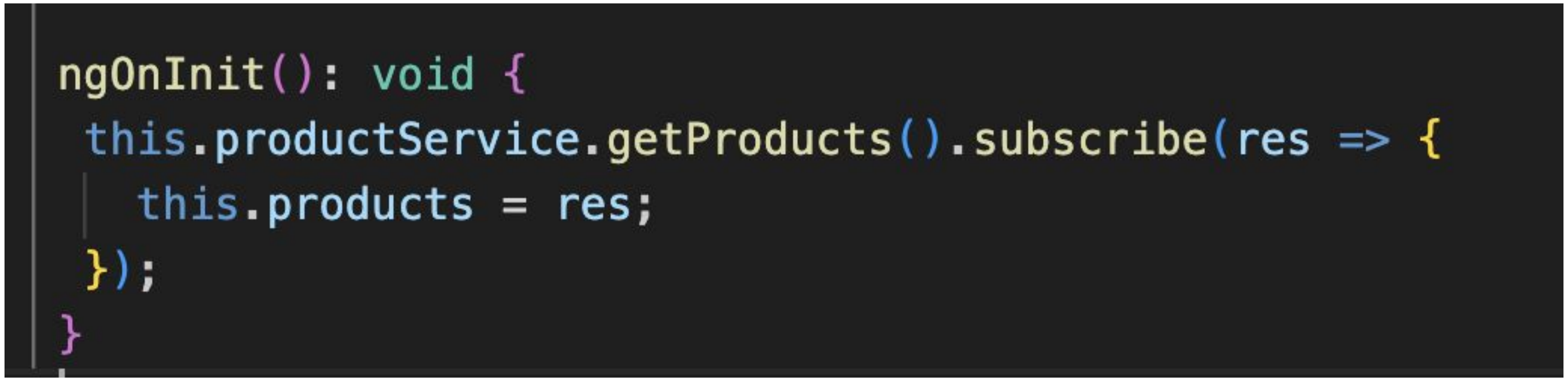
```
  getProducts(): Observable<Product[]> {
    return this.http.get<ProductDTO[]>(this.productsUrl).pipe(
    map(products => products.map(product => {
    return {
    name: product.title,
    price: product.price
    }
    }))
    );
}
```

Updated code with comment and more fields based on interface

```
getProducts():Observable<Product[]>{
   // Call the API and get a response of type Array of product
   return this.httpClient.get<Product[]>(this.url).pipe(
     // Transform it into observable
     map(
       // for each of the response
       // I will transform the response into product object (map)
       response => response.map(val =>{
         return {
           id:val.id,
           title:val.title,
           price:val.price,
           image:val.image
         }
```

Fetching data through HTTP (3)

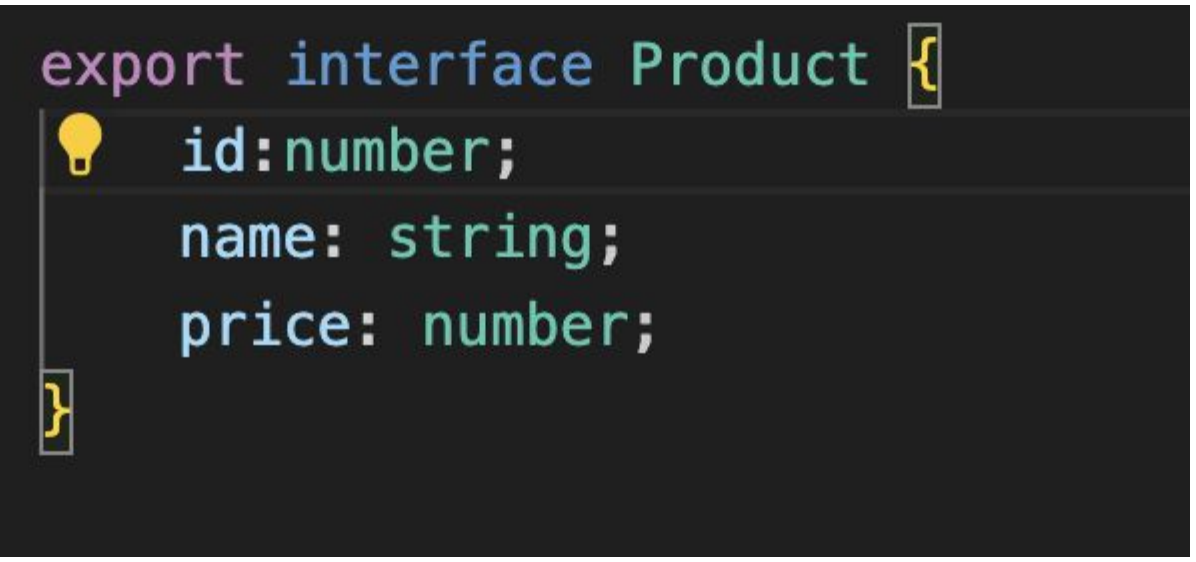
product-list.component.ts

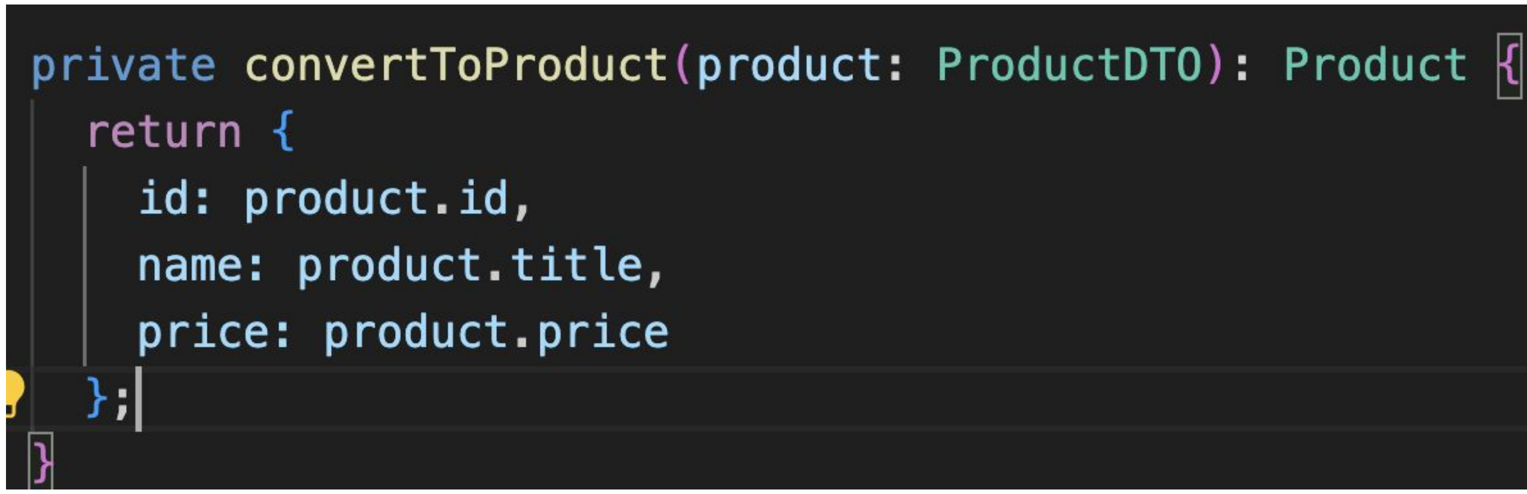


```
ngOnInit(): void {
 this.productService.getProducts().subscribe(res => {
   this.products = res;
 });
}
```

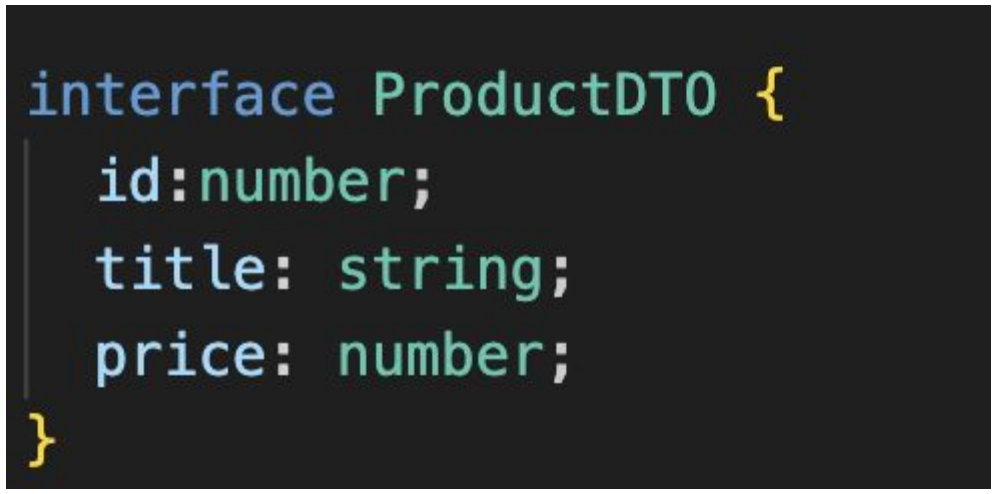
Fetch data by Id through HTTP (1)

product.ts


```
export interface Product {
    id:number;
    name: string;
    price: number;
}
```

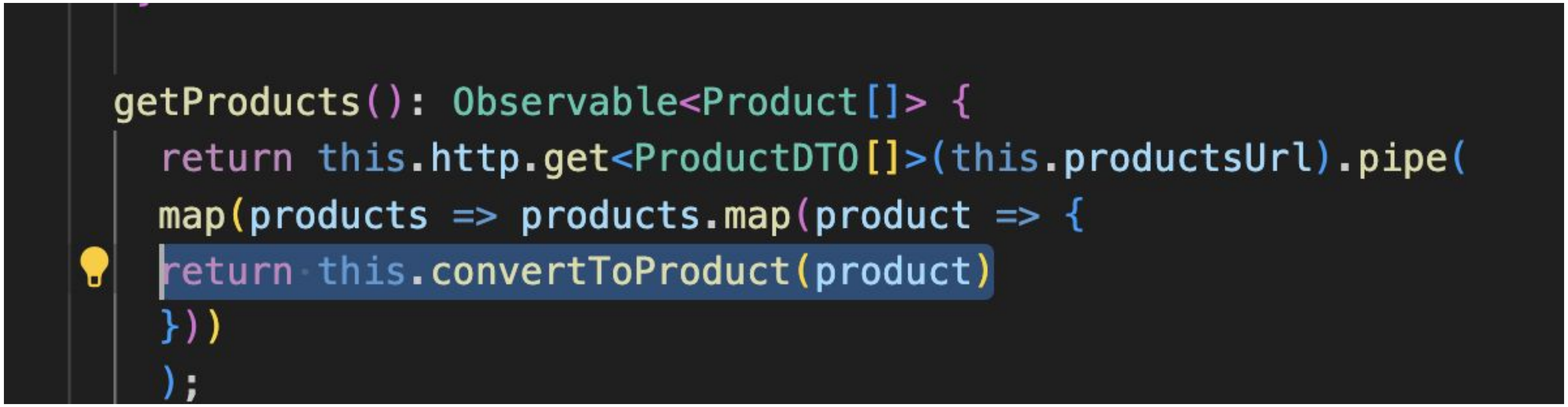

```
private convertToProduct(product: ProductDTO): Product {
  return {
    id: product.id,
    name: product.title,
    price: product.price
  };
}
```

product.service.ts

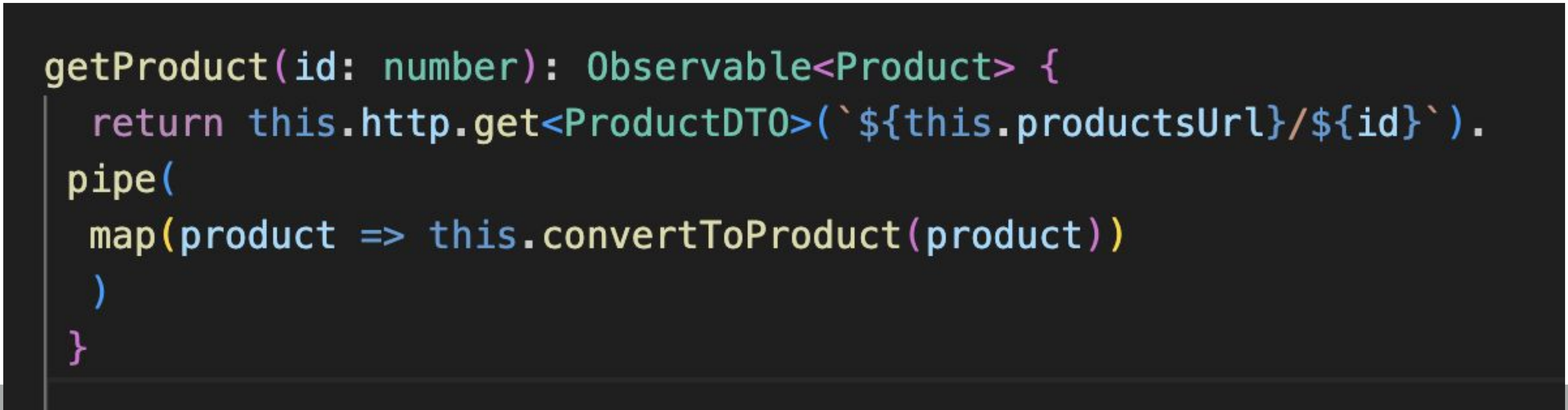

```
interface ProductDTO {
  id:number;
  title: string;
  price: number;
}
```

Fetch data by Id through HTTP (2)

product.service.ts



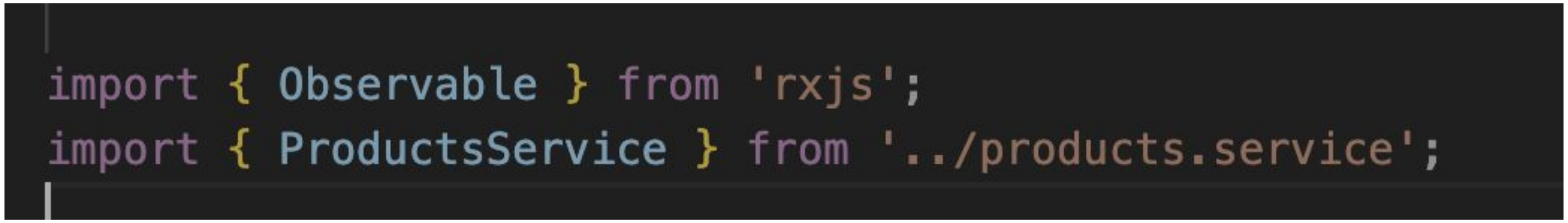
```
getProducts(): Observable<Product[]> {
  return this.http.get<ProductDTO[]>(this.productsUrl).pipe(
  map(products => products.map(product => {
  return this.convertToProduct(product)
  }))
  );
```



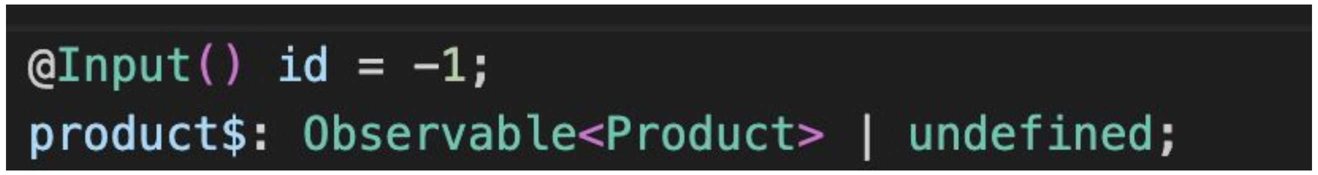
```
getProduct(id: number): Observable<Product> {
  return this.http.get<ProductDTO>(`${this.productsUrl}/${id}`).
 pipe(
  map(product => this.convertToProduct(product))
  )
 }
```

Fetch data by Id through HTTP (3)

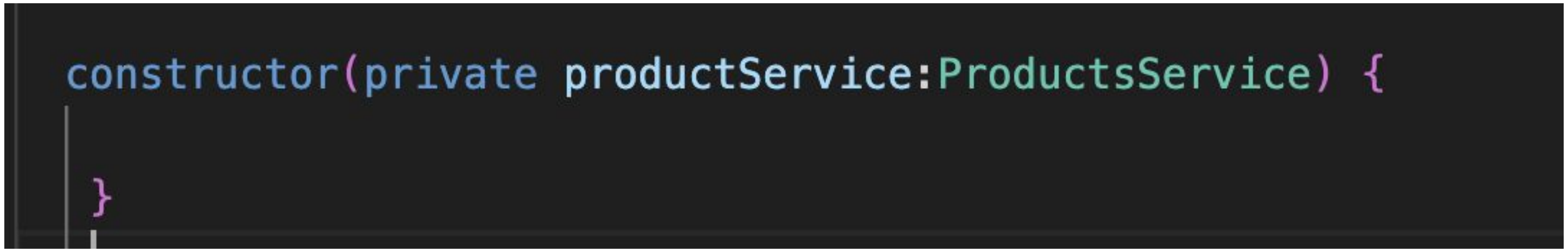
product-detail.component.ts



```
import { Observable } from 'rxjs';
import { ProductsService } from '../products.service';
```



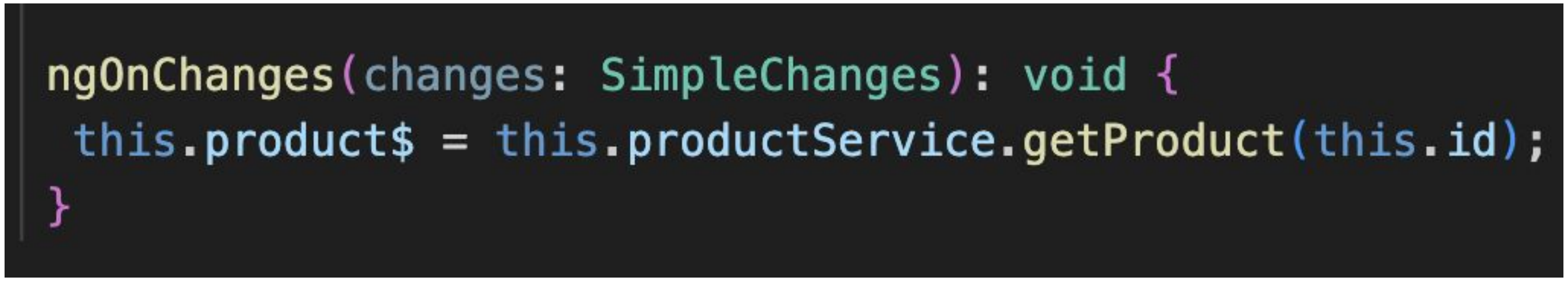
```
@Input() id = -1;
product$: Observable<Product> | undefined;
```



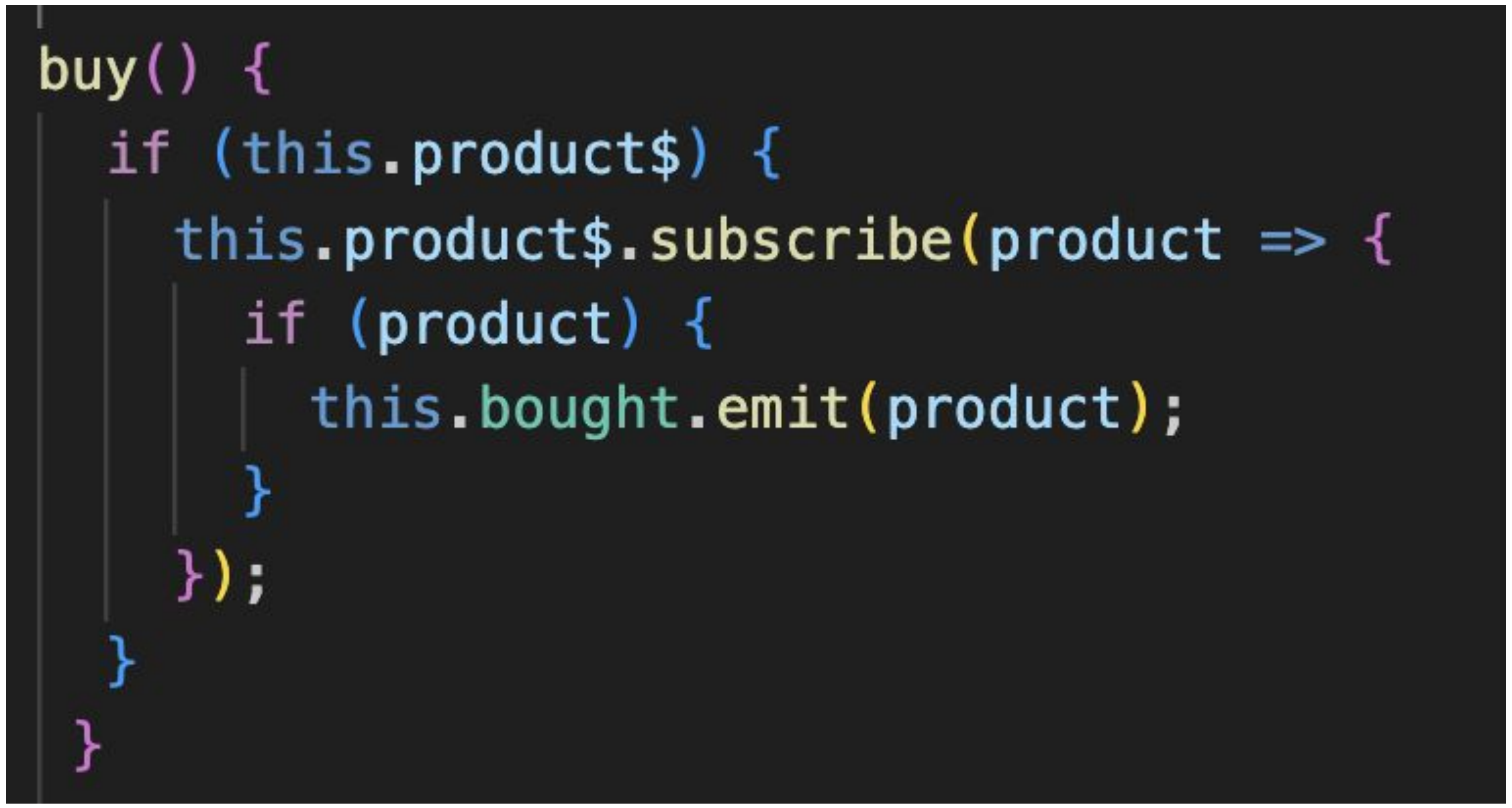
```
constructor(private productService:ProductsService) {

 }
```

Fetch data by Id through HTTP (4)

product-detail.component.ts



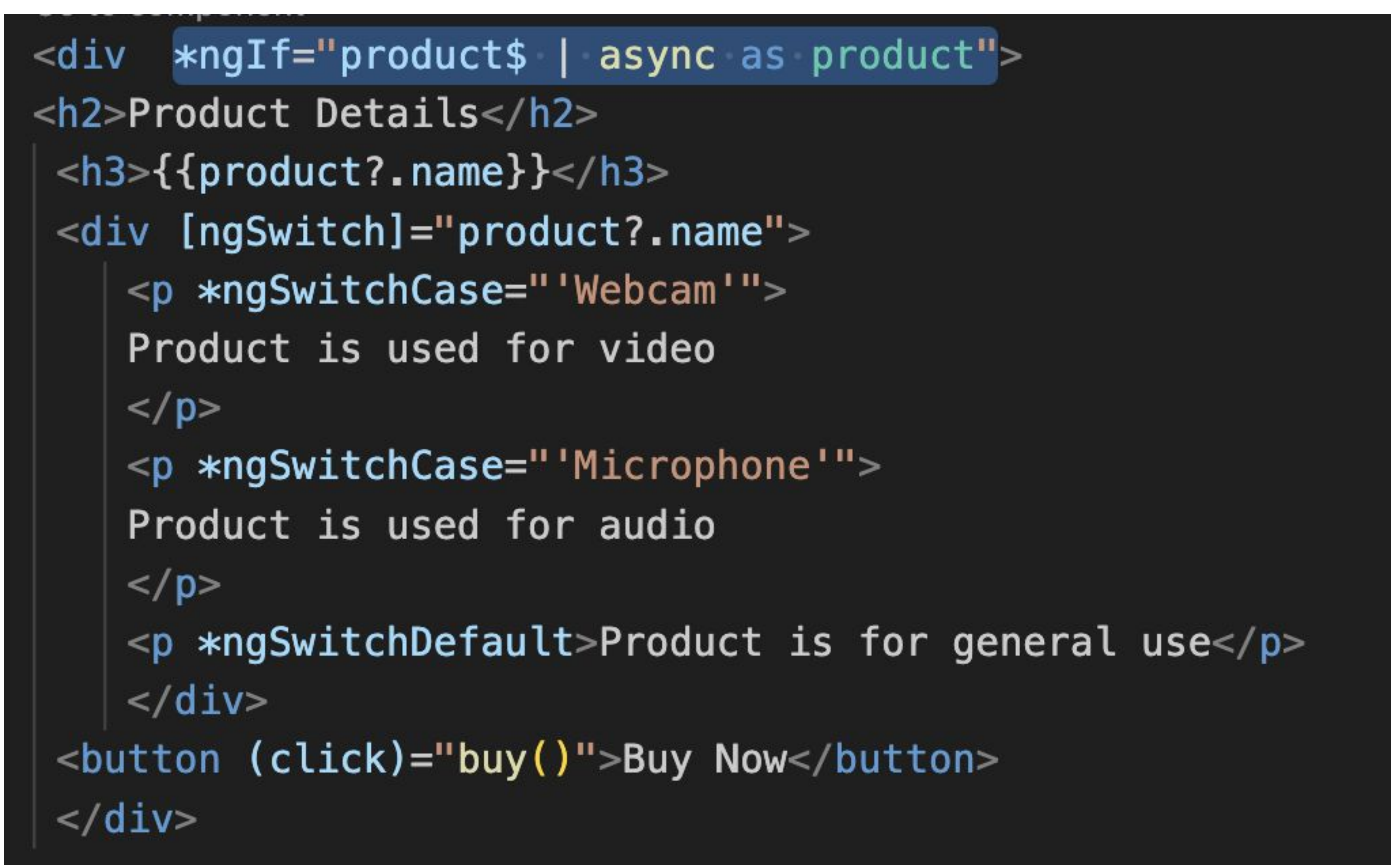
```
ngOnChanges(changes: SimpleChanges): void {
 this.product$ = this.productService.getProduct(this.id);
}
```



```
buy() {
  if (this.product$) {
    this.product$.subscribe(product => {
      if (product) {
        this.bought.emit(product);
      }
    });
  }
}
```

Fetch data by Id through HTTP (5)

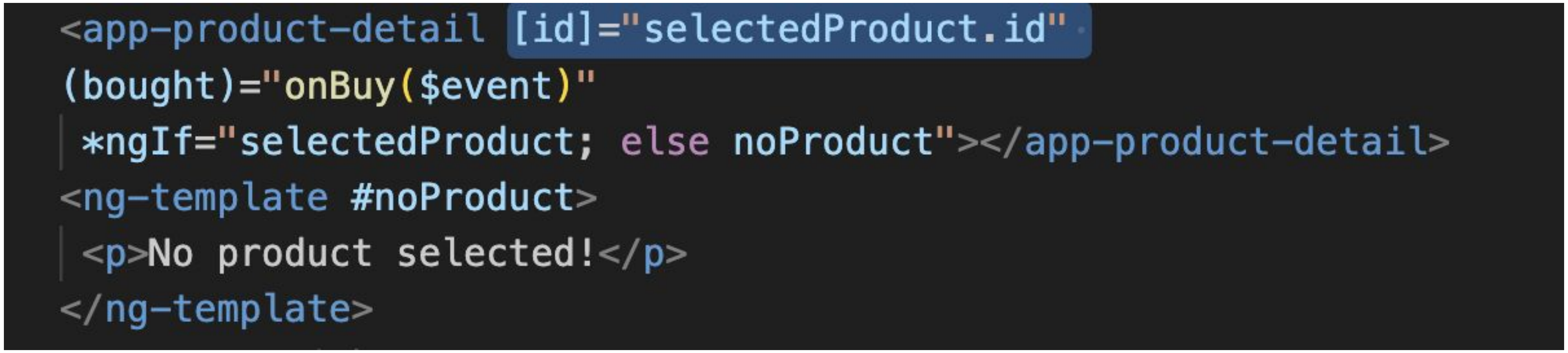
product-detail.component.ts



```
<div  *ngIf="product$ | async as product">
<h2>Product Details</h2>
 <h3>{{product?.name}}</h3>
 <div [ngSwitch]="product?.name">
    <p *ngSwitchCase="'Webcam'">
    Product is used for video
    </p>
    <p *ngSwitchCase="'Microphone'">
    Product is used for audio
    </p>
    <p *ngSwitchDefault>Product is for general use</p>
    </div>
 <button (click)="buy()">Buy Now</button>
 </div>
```

Fetch data by Id through HTTP (6)

product-list.component.html



```
<app-product-detail [id]="selectedProduct.id"
(bought)="onBuy($event)"
 *ngIf="selectedProduct; else noProduct"></app-product-detail>
<ng-template #noProduct>
 <p>No product selected!</p>
</ng-template>
```

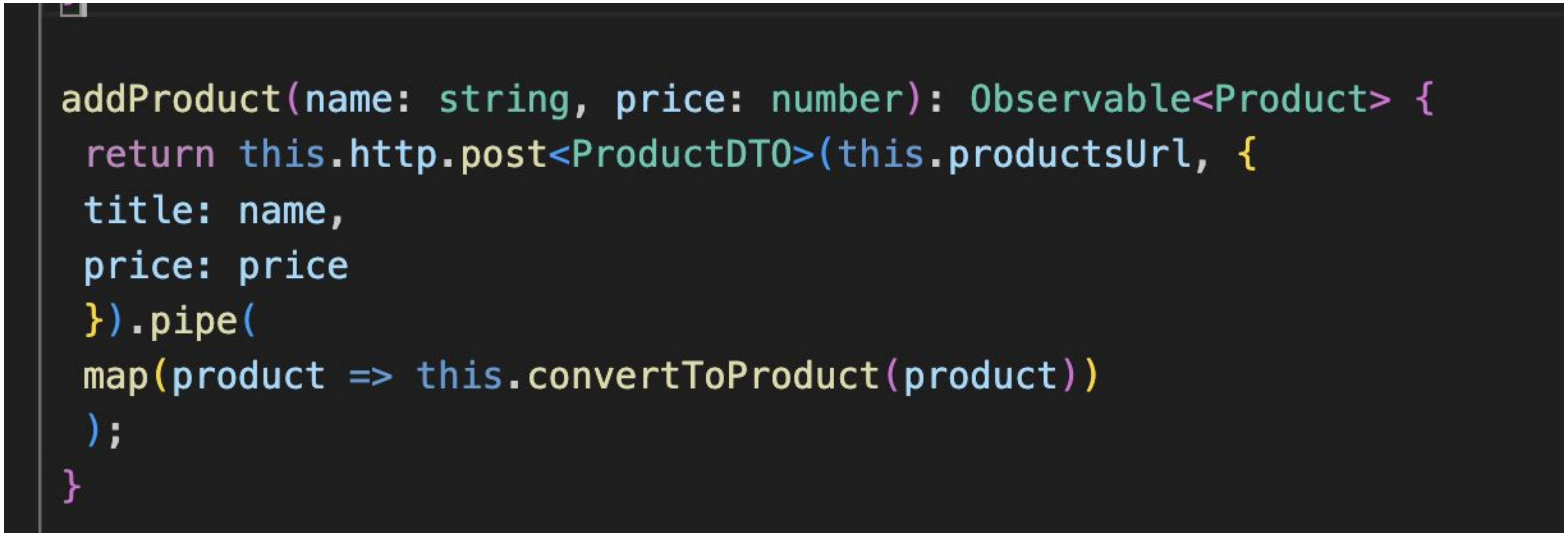
Modifying Data through HTTP

Modifying data in a CRUD application usually refers to adding new data and updating or deleting existing data. To demonstrate how to implement such functionality in an Angular application using the HttpClient service, we will make the following changes to our application:

- Create an Angular component to add new products.
- Modify the product detail component to change the price of an existing product.
- Add a button in the product detail component for deleting an existing product.

Add new product

product.service.ts



```
addProduct(name: string, price: number): Observable<Product> {
 return this.http.post<ProductDTO>(this.productsUrl, {
 title: name,
 price: price
 }).pipe(
 map(product => this.convertToProduct(product))
 );
}
```

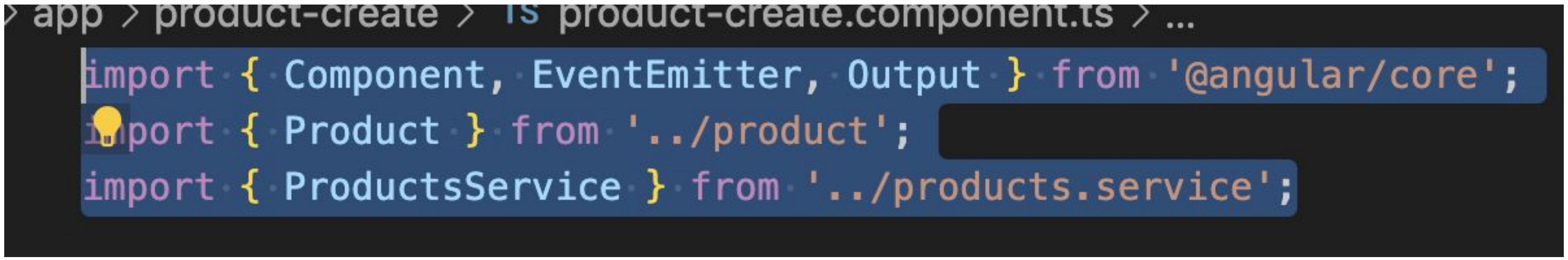
Add new product (2)

Generate new component:

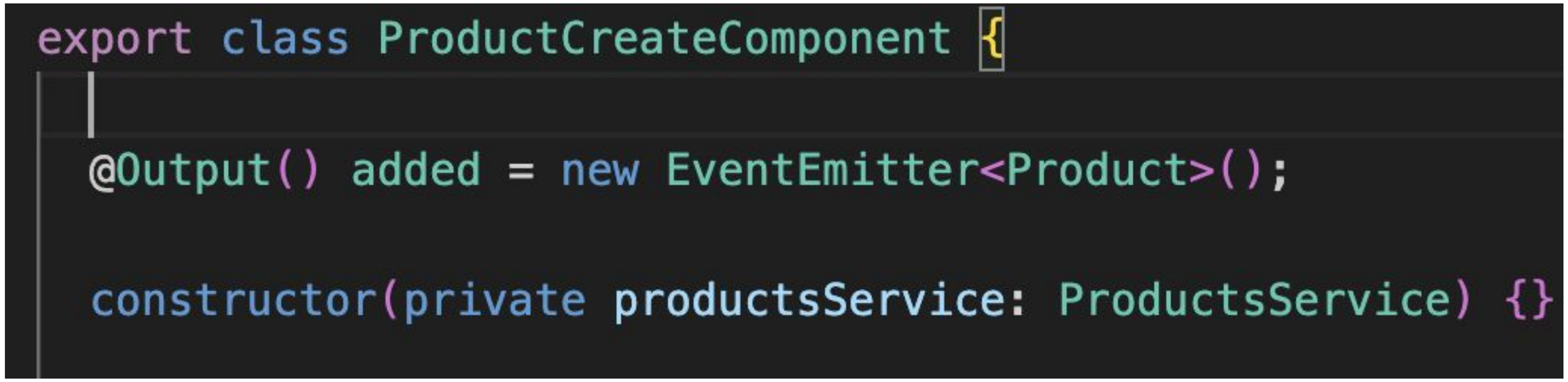
```
ng generate component product-create
```

Add new product (3)

product-create.component.ts



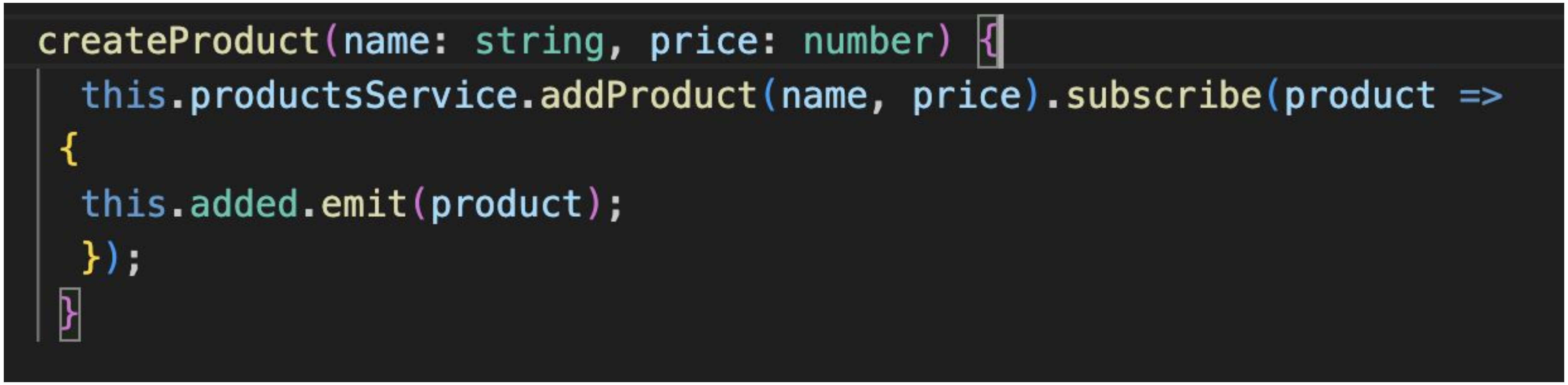
```
> app > product-create > TS product-create.component.ts > ...
import { Component, EventEmitter, Output } from '@angular/core';
import { Product } from '../product';
import { ProductsService } from '../products.service';
```



```
export class ProductCreateComponent {

  @Output() added = new EventEmitter<Product>();

  constructor(private productsService: ProductsService) {}
```

Add new product (4)

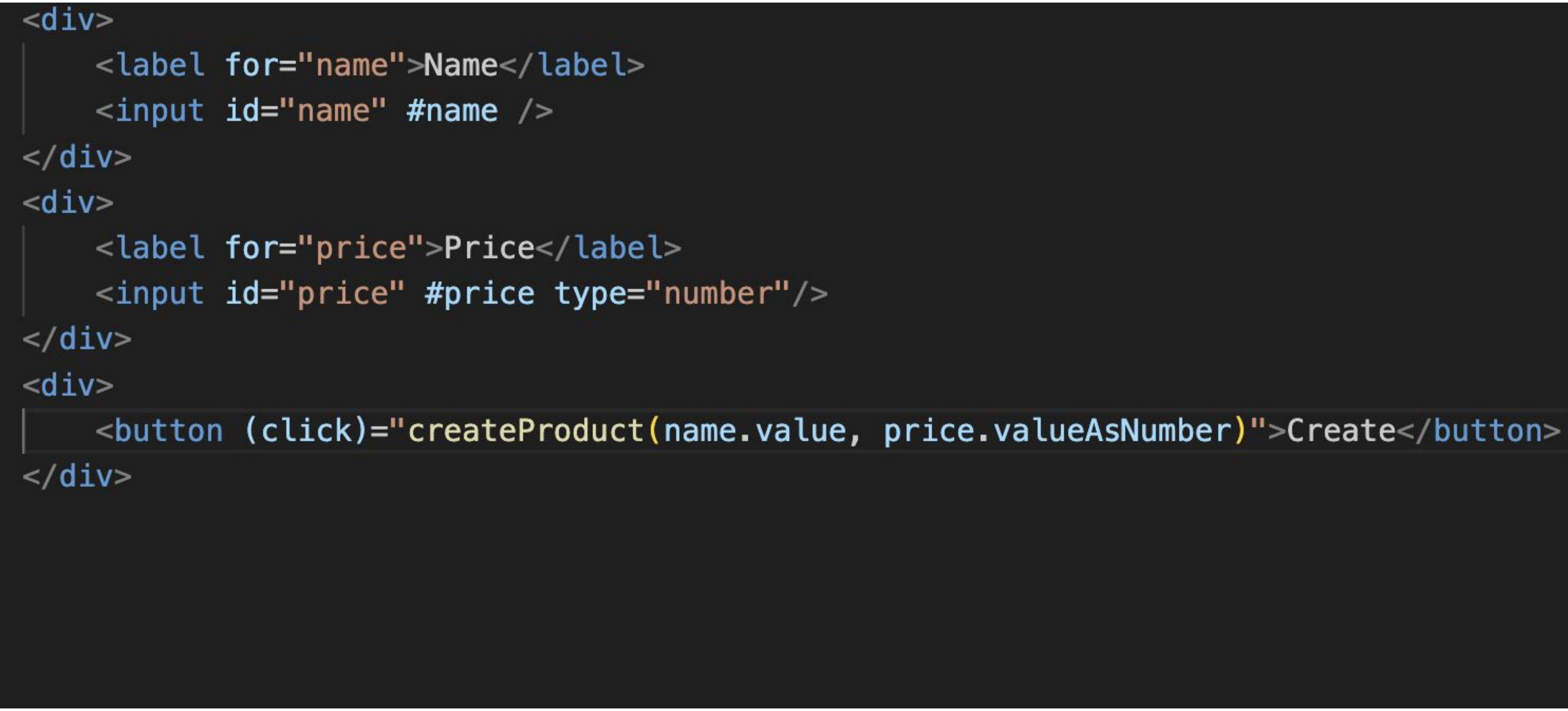
product-create.component.ts



```
createProduct(name: string, price: number) {
   this.productsService.addProduct(name, price).subscribe(product =>
  {
   this.added.emit(product);
   });
  }
```

Add new product (5)

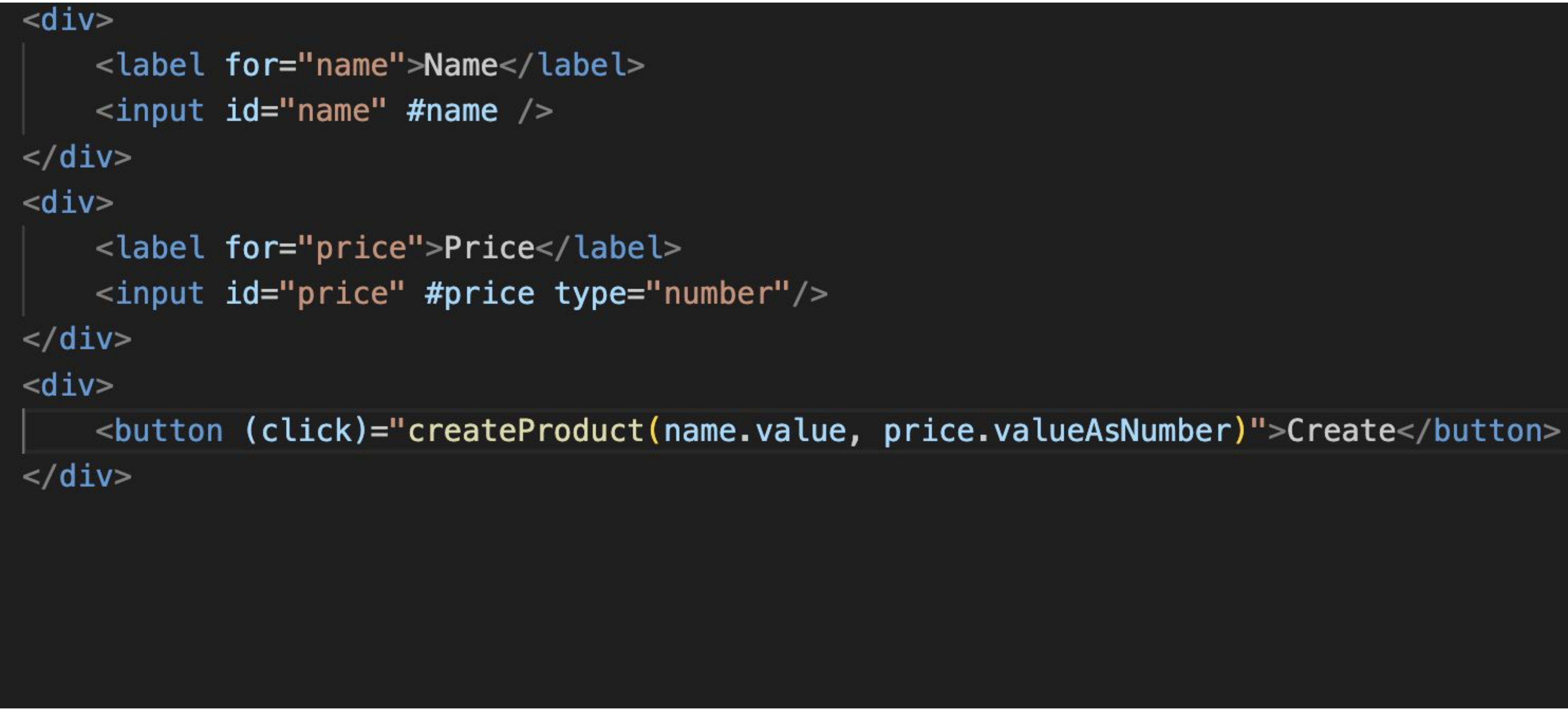
product-create.component.html



```
<div>
    <label for="name">Name</label>
    <input id="name" #name />
</div>
<div>
    <label for="price">Price</label>
    <input id="price" #price type="number"/>
</div>
<div>
    <button (click)="createProduct(name.value, price.valueAsNumber)">Create</button>
</div>
```

Add new product (5)

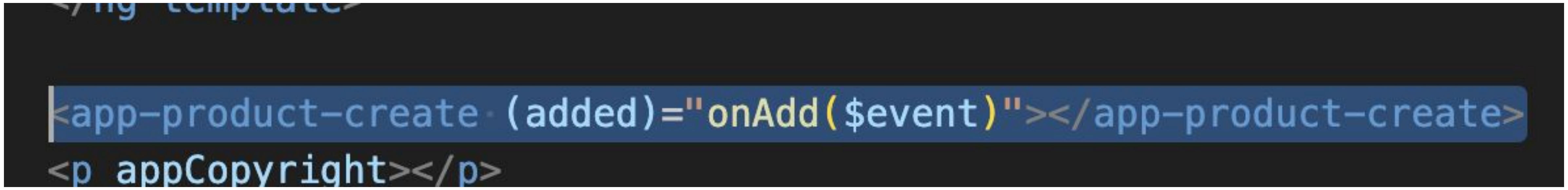
product-create.component.html



```html
<div>
    <label for="name">Name</label>
    <input id="name" #name />
</div>
<div>
    <label for="price">Price</label>
    <input id="price" #price type="number"/>
</div>
<div>
    <button (click)="createProduct(name.value, price.valueAsNumber)">Create</button>
</div>
```

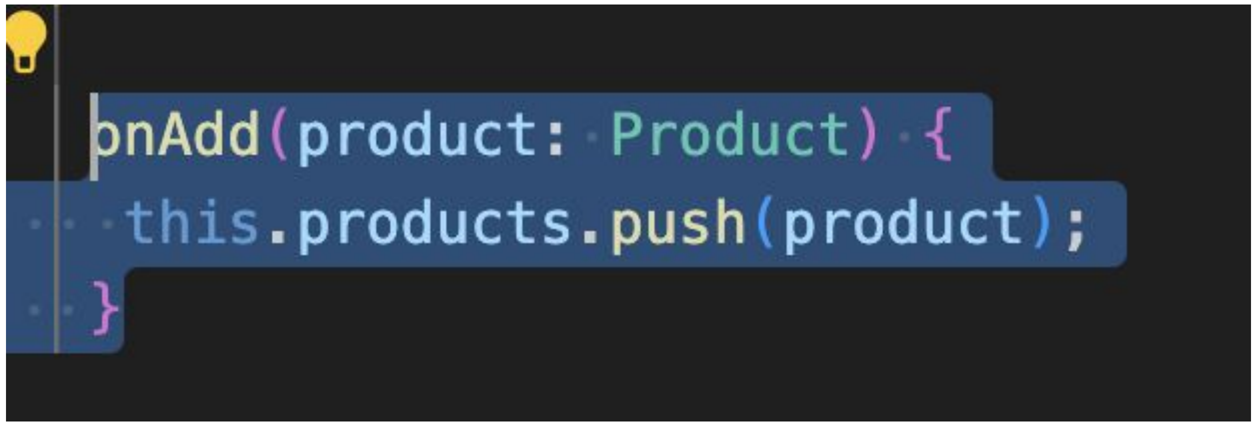
Add new product (6)

Product-list.component.html



```html
<app-product-create (added)="onAdd($event)"></app-product-create>
<p appCopyright></p>
```

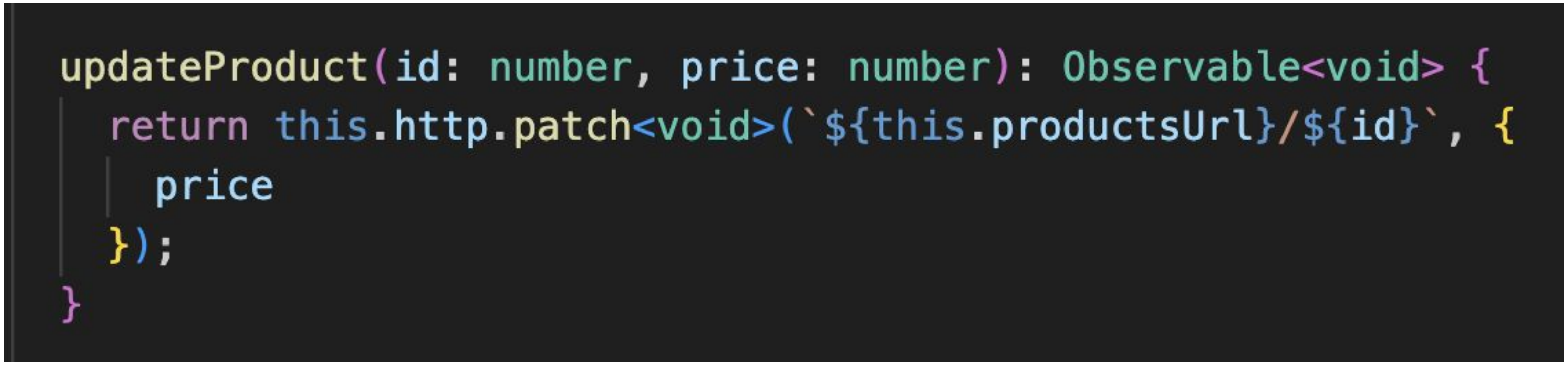
product-list.component.ts



```typescript
onAdd(product: Product) {
  this.products.push(product);
}
```

Updating product price

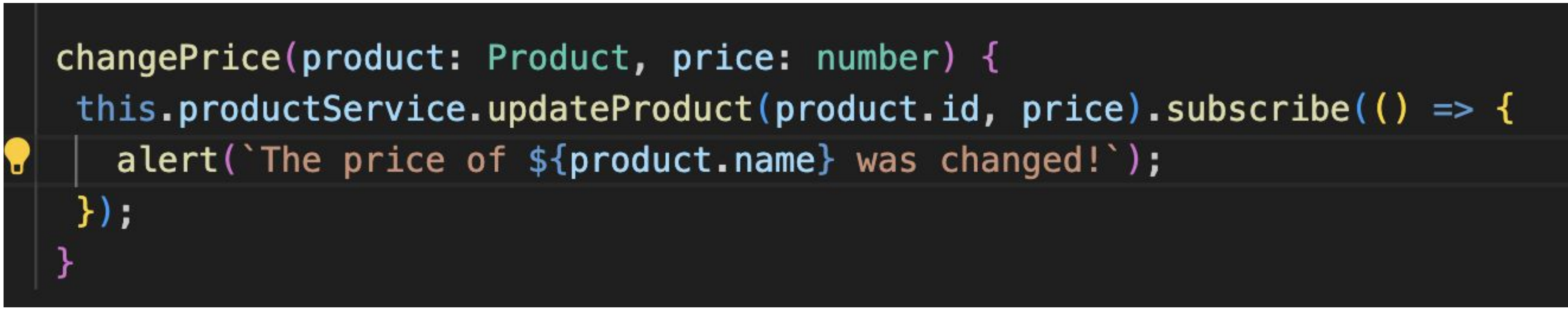
product.service.ts



```
updateProduct(id: number, price: number): Observable<void> {
  return this.http.patch<void>(`${this.productsUrl}/${id}`, {
    price
  });
}
```

Updating product price (2)

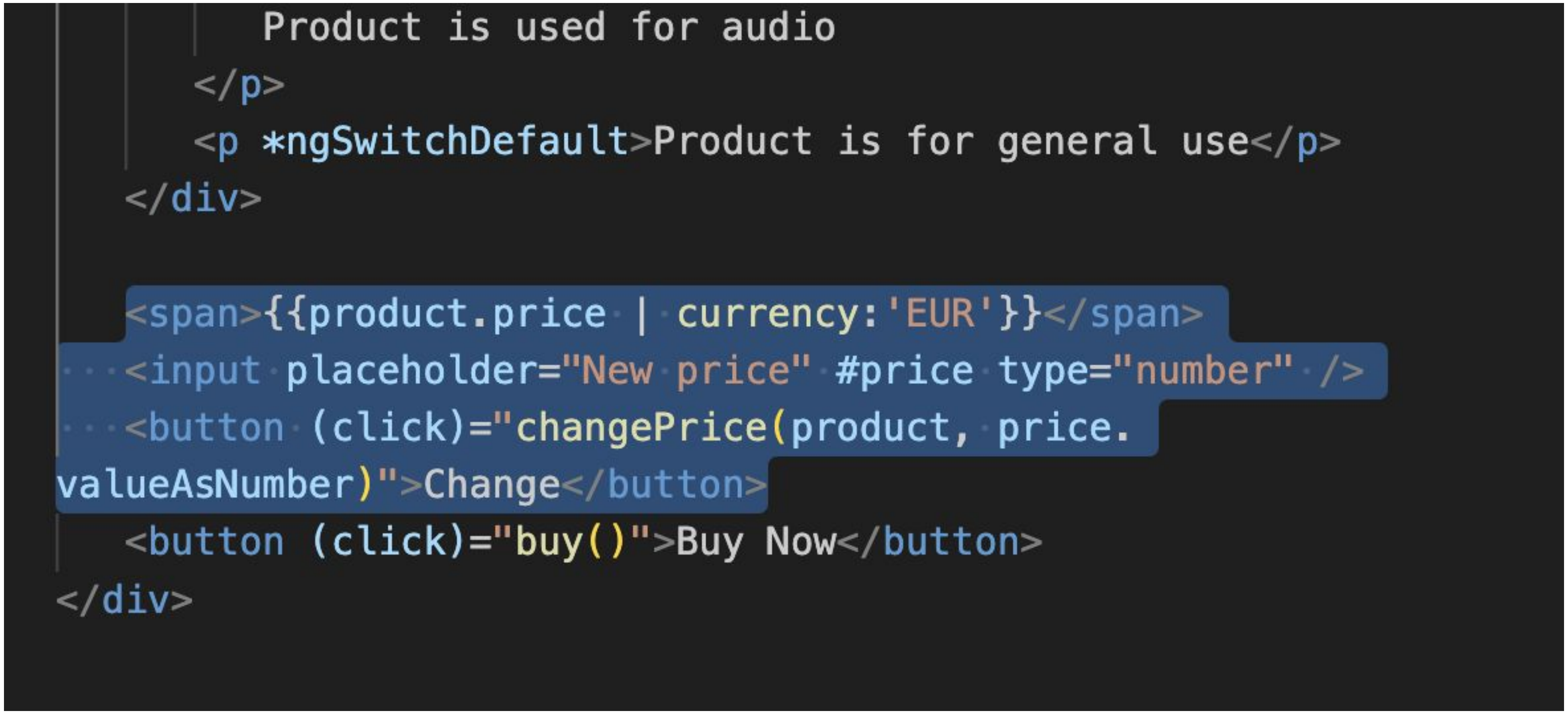
product-detail.component.ts



```
changePrice(product: Product, price: number) {
 this.productService.updateProduct(product.id, price).subscribe(() => {
   alert(`The price of ${product.name} was changed!`);
 });
}
```

Updating product price (3)

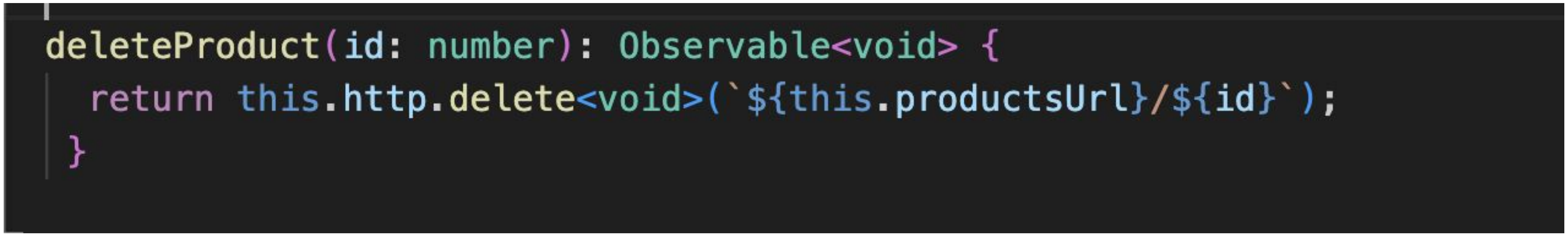
product-detail.component.ts



```
          Product is used for audio
        </p>
        <p *ngSwitchDefault>Product is for general use</p>
    </div>

    <span>{{product.price | currency:'EUR'}}</span>
    <input placeholder="New price" #price type="number" />
    <button (click)="changePrice(product, price.
valueAsNumber)">Change</button>
    <button (click)="buy()">Buy Now</button>
</div>
```

Removing a product (1)

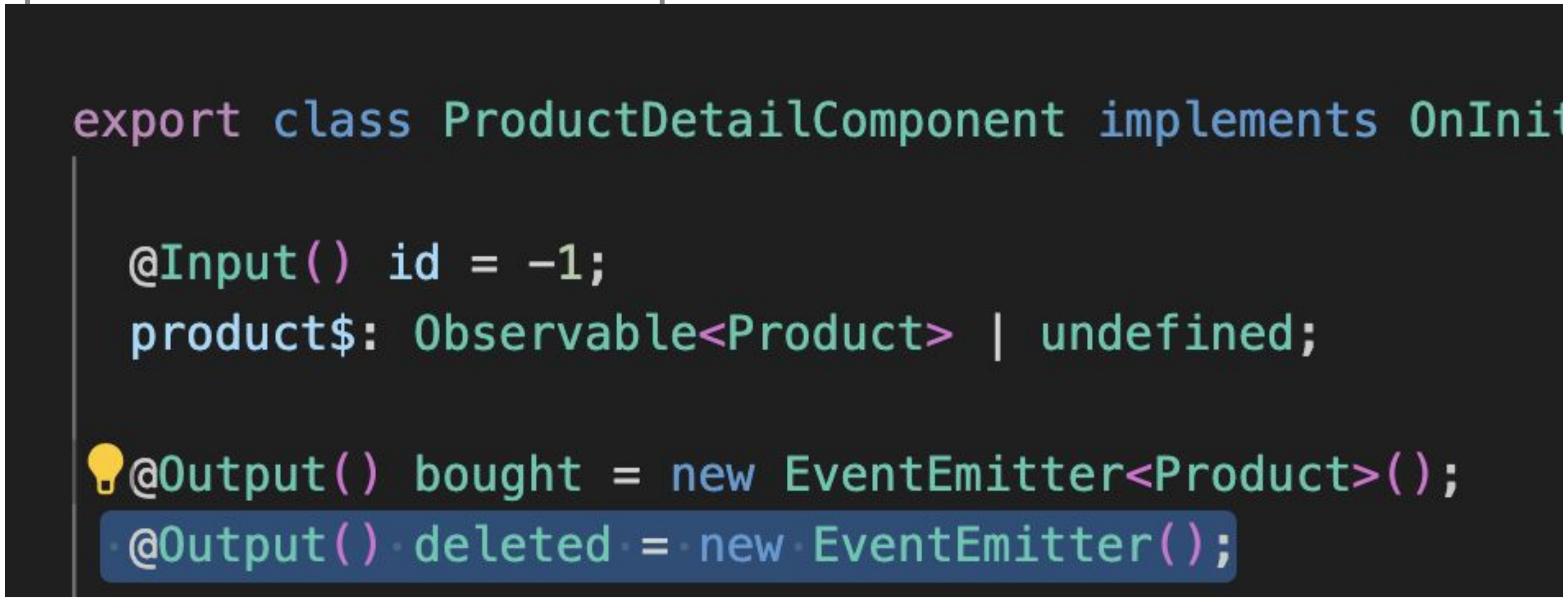
product.service.ts



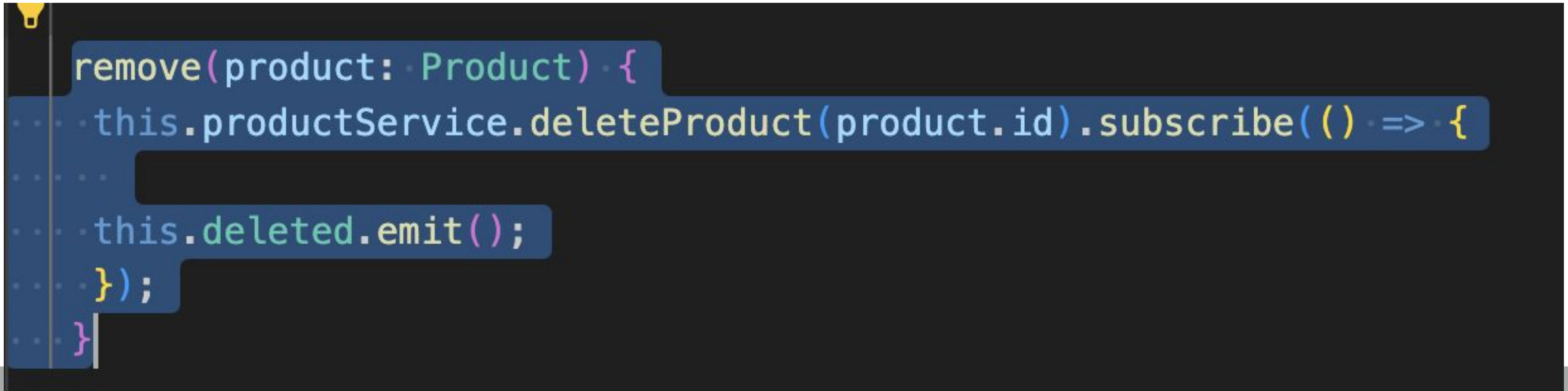
```
deleteProduct(id: number): Observable<void> {
  return this.http.delete<void>(`${this.productsUrl}/${id}`);
}
```

Removing a product (1)

product-detail.component.ts



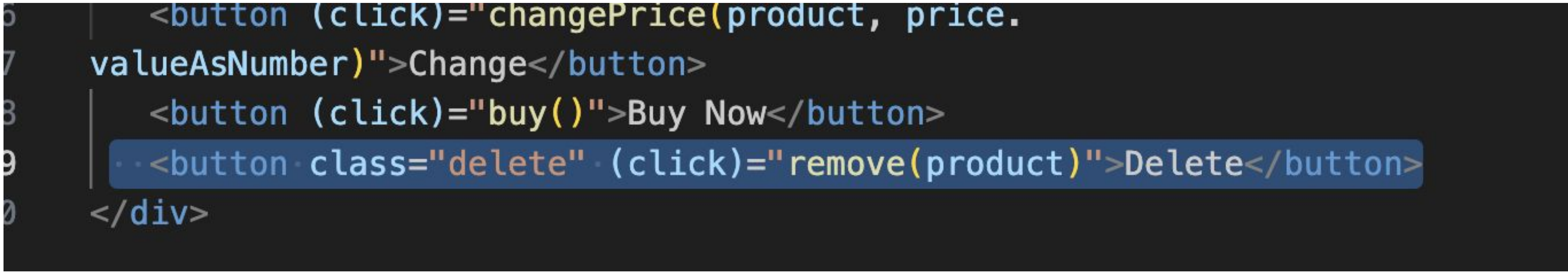
```
export class ProductDetailComponent implements OnInit

  @Input() id = -1;
  product$: Observable<Product> | undefined;

  @Output() bought = new EventEmitter<Product>();
  @Output() deleted = new EventEmitter();
```



```
remove(product: Product) {
  this.productService.deleteProduct(product.id).subscribe(() => {

  this.deleted.emit();
  });
}
```

Removing a product (2)

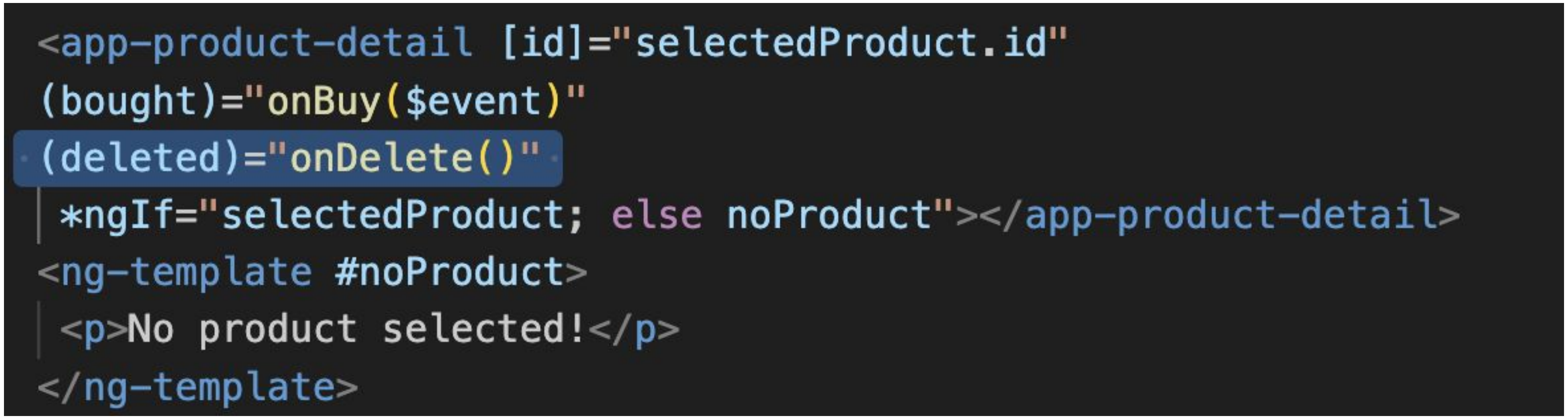
product-detail.component.html



```
    <button (click)="changePrice(product, price.
valueAsNumber)">Change</button>
    <button (click)="buy()">Buy Now</button>
    <button class="delete" (click)="remove(product)">Delete</button>
</div>
```

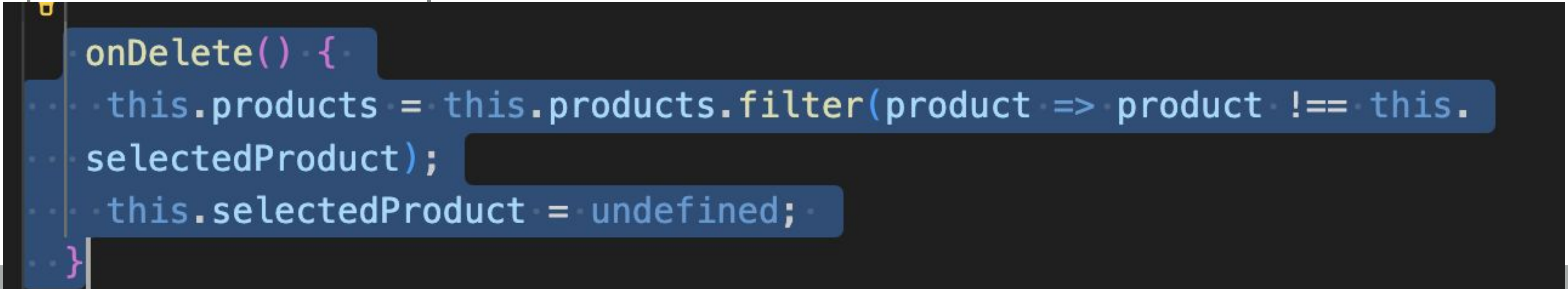
Removing a product (3)

product-list.component.html



```
<app-product-detail [id]="selectedProduct.id"
(bought)="onBuy($event)"
(deleted)="onDelete()"
 *ngIf="selectedProduct; else noProduct"></app-product-detail>
<ng-template #noProduct>
 <p>No product selected!</p>
</ng-template>
```

product-list.component.ts



```
  onDelete() {
    this.products = this.products.filter(product => product !== this.
  selectedProduct);
    this.selectedProduct = undefined;
  }
```

Authentication and Authorization

We will explore the following authentication and authorization topics in this section using the fakestoreapi.com

- Authenticating with a backend API
- Authorizing users for certain features
- Authorizing HTTP requests using interceptors

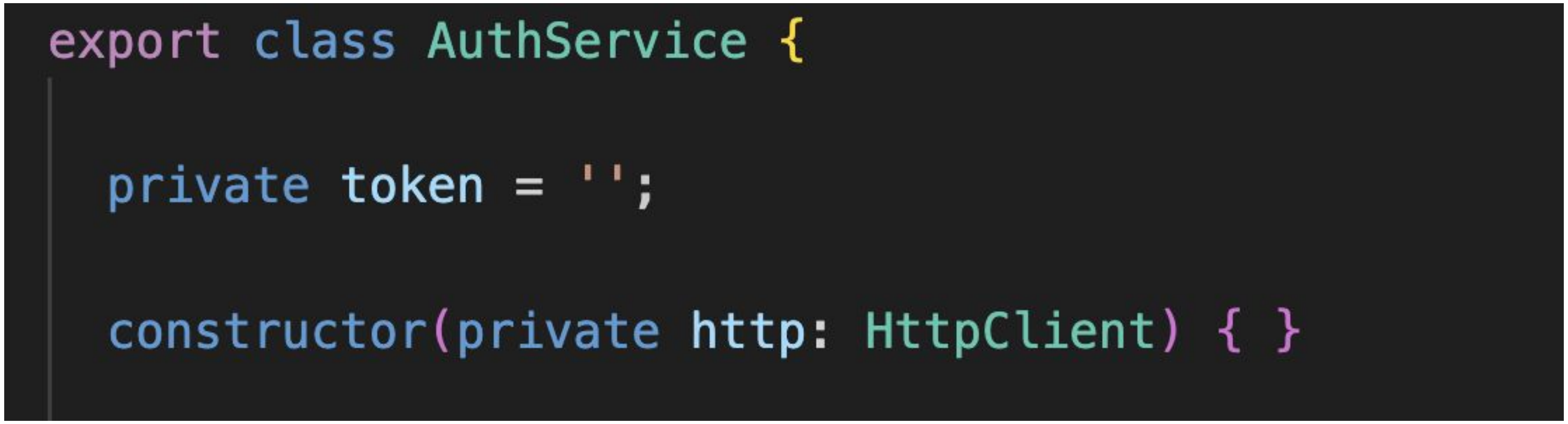
Authentication and Authorization (2)

Generate new service that will handle the authentication and authorization:

```
ng generate service auth
```

Authentication and Authorization (3)

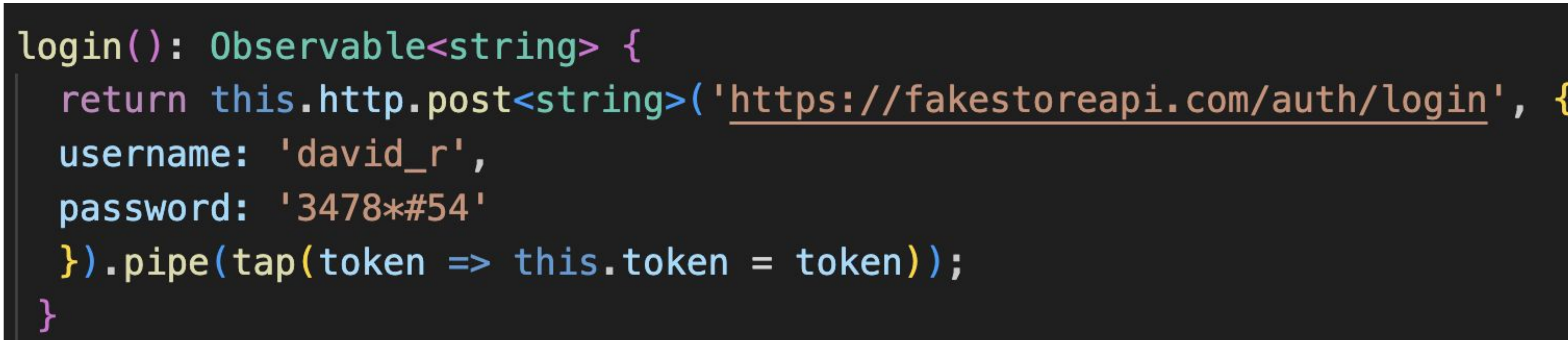
Create a private property token and inject HttpClient inside it



```
export class AuthService {

  private token = '';

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

Authentication and Authorization (4)

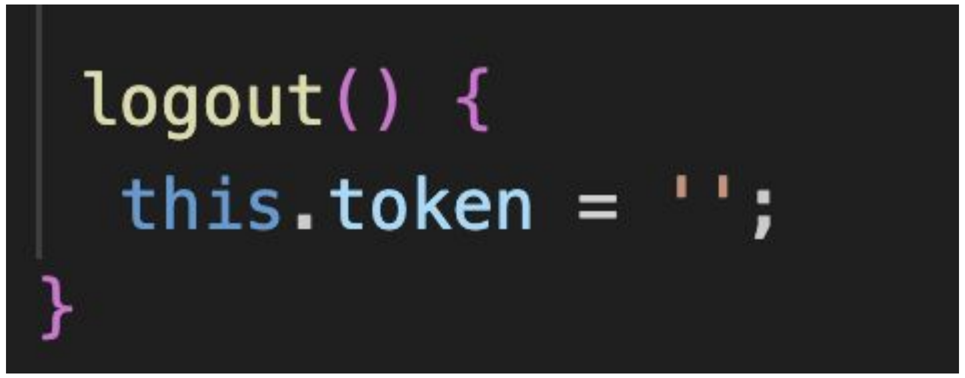
Create a login method in the service



```
login(): Observable<string> {
  return this.http.post<string>('https://fakestoreapi.com/auth/login', {
  username: 'david_r',
  password: '3478*#54'
  }).pipe(tap(token => this.token = token));
 }
```

Authentication and Authorization (5)

Create a logout method on the token



```
  logout() {
   this.token = '';
}
```

Authentication and Authorization (6)

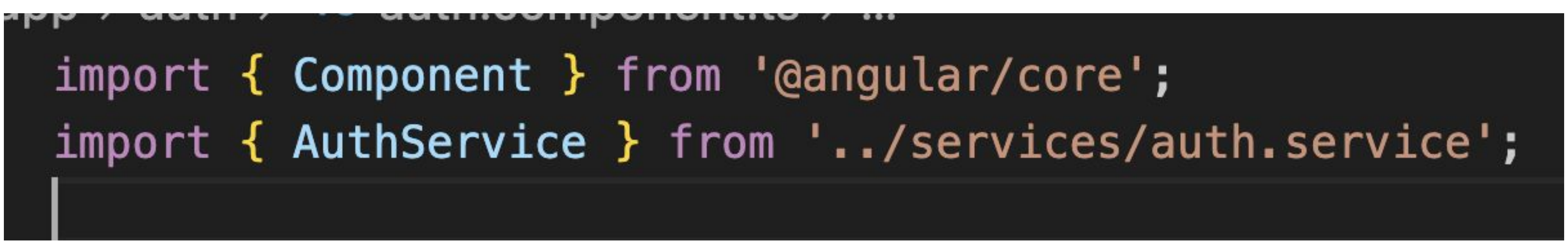
Create a new auth component to simulate login and logout

```
ng generate component auth
```

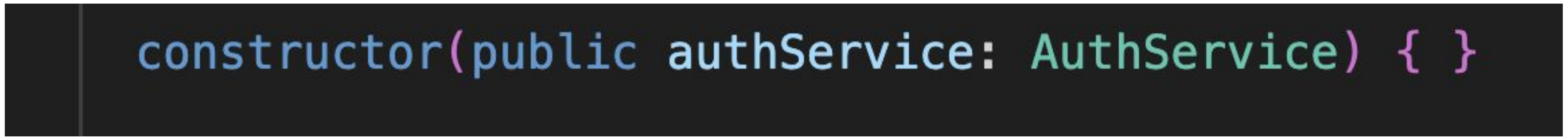
Authentication and Authorization (6)

Import and inject the auth.service inside the component:

auth.component.ts



```
import { Component } from '@angular/core';
import { AuthService } from '../services/auth.service';
```

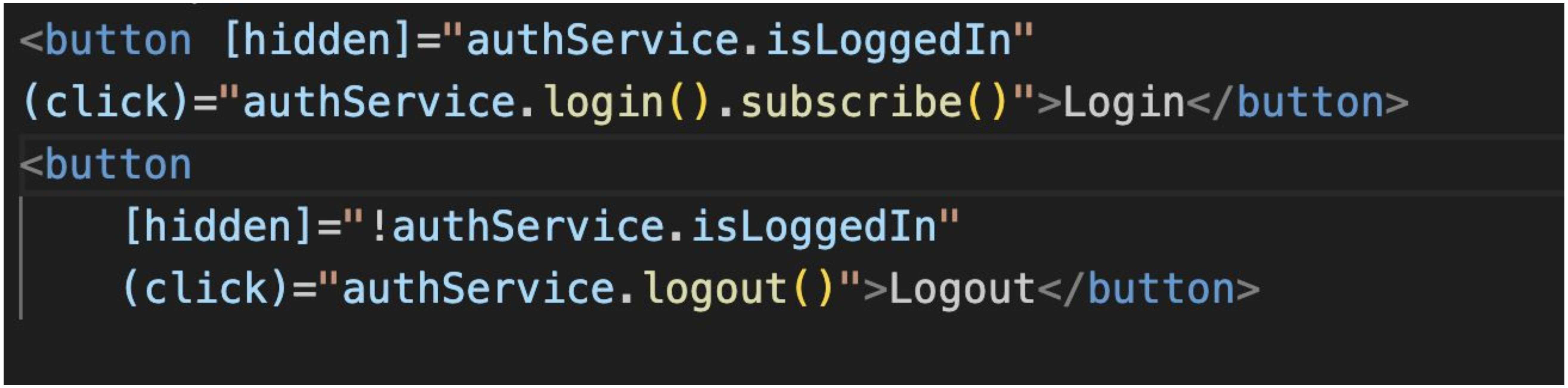


```
constructor(public authService: AuthService) { }
```

Authentication and Authorization (7)

Create the UI for the login and logout

auth.component.html

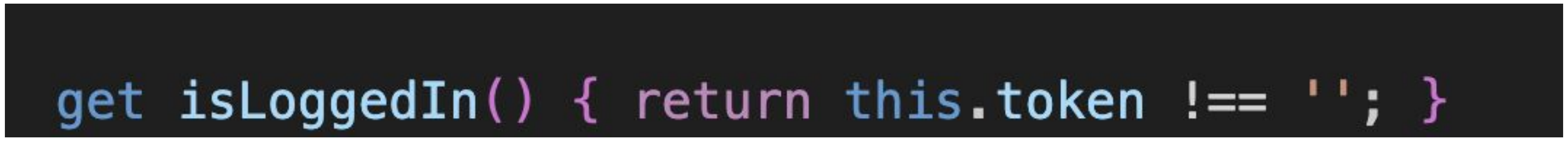


```
<button [hidden]="authService.isLoggedIn"
(click)="authService.login().subscribe()">Login</button>
<button
    [hidden]="!authService.isLoggedIn"
    (click)="authService.logout()">Logout</button>
```

Authentication and Authorization (8)

Create a getter to verify the token existence in the service

auth.service.ts

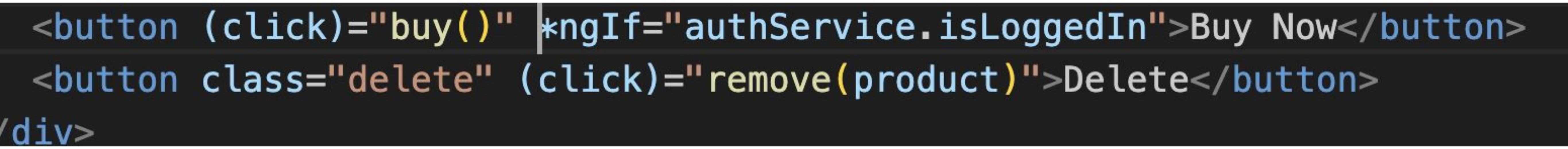


```
get isLoggedIn() { return this.token !== ''; }
```

Authentication and Authorization (9)

In product detail, simulate a “only logged in scenario”, for example the buy button can only be pressed after log in

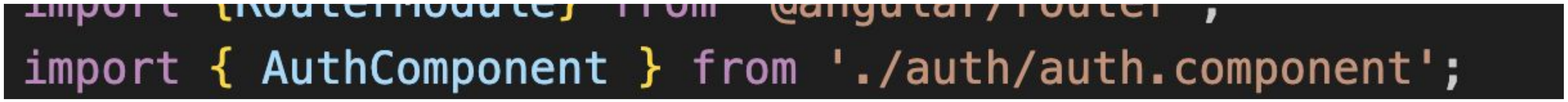
product-detail.component.html



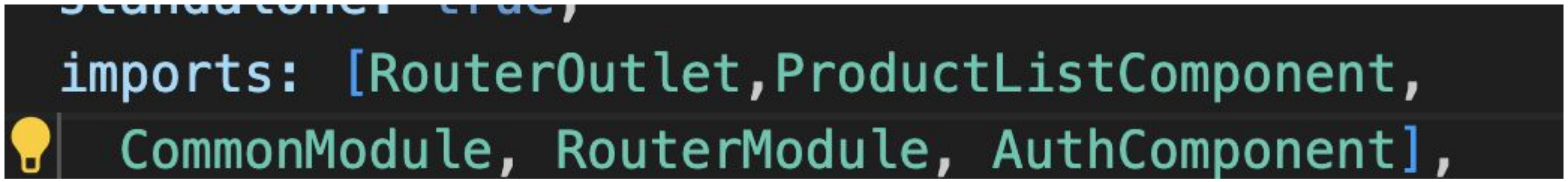
```
  <button (click)="buy()" *ngIf="authService.isLoggedIn">Buy Now</button>
  <button class="delete" (click)="remove(product)">Delete</button>
/div>
```

Authentication and Authorization (10)

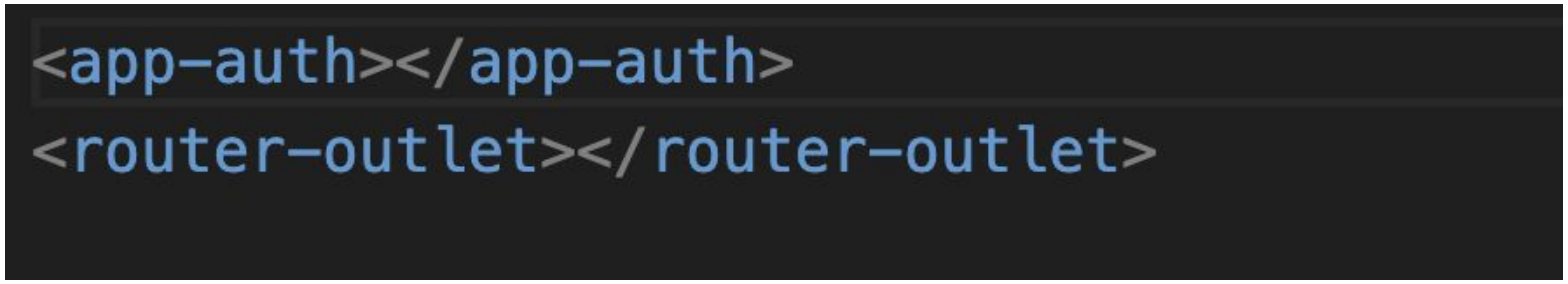
Add the AuthComponent on app.component.html



```
import {RouterModule} from '@angular/router';
import { AuthComponent } from './auth/auth.component';
```



```
  standalone: true,
  imports: [RouterOutlet,ProductListComponent,
    CommonModule, RouterModule, AuthComponent],
```

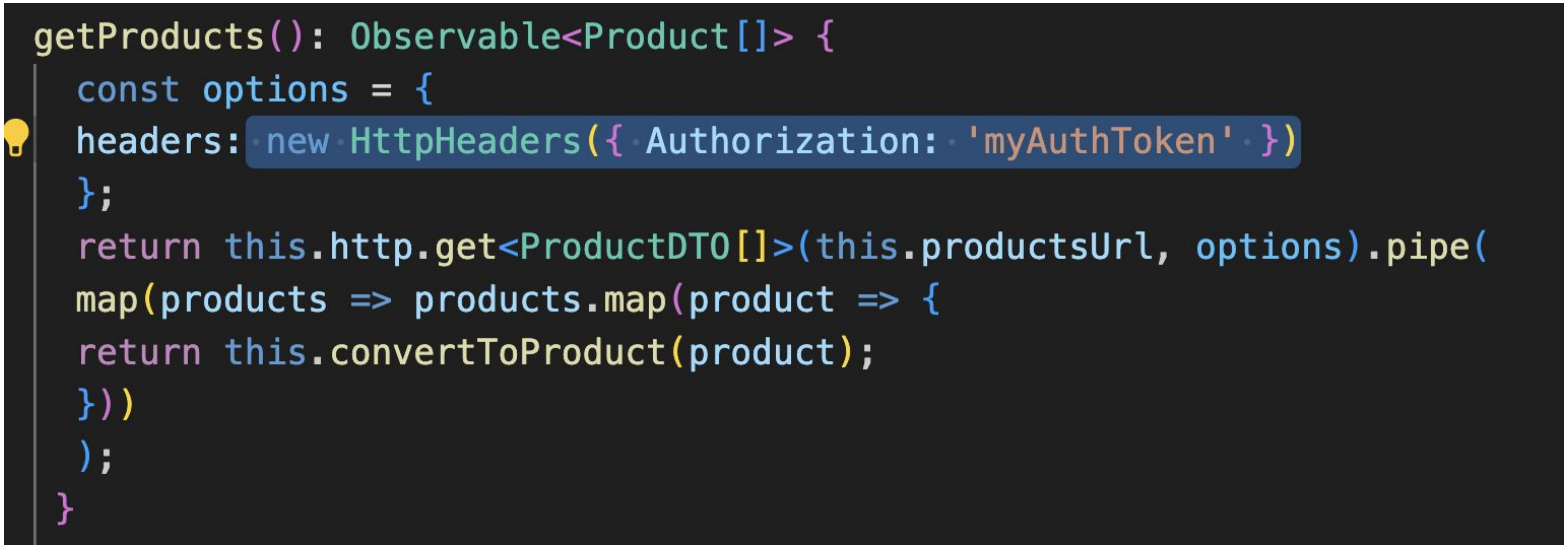


```
<app-auth></app-auth>
<router-outlet></router-outlet>
```

Authorizing Http Request

A common pattern in web applications is to include the token in an Authorization header.

We can use HTTP headers in an Angular application by importing the HttpHeaders artifact from the @angular/common/http namespace and modifying our methods accordingly:



```
getProducts(): Observable<Product[]> {
  const options = {
  headers: new HttpHeaders({ Authorization: 'myAuthToken' })
  };
  return this.http.get<ProductDTO[]>(this.productsUrl, options).pipe(
  map(products => products.map(product => {
  return this.convertToProduct(product);
  }))
  );
 }
```

Interceptor

An HTTP interceptor is an Angular service that intercepts HTTP requests and responses that pass through the Angular built-in HTTP client. It can be used in the following scenarios:

- When we want to pass custom HTTP headers in every request, such as an authentication token.

- When we want to display a loading indicator while we wait for a response from the server.

- When we want to provide a logging mechanism for every HTTP communication.

https://angular.io/guide/http-interceptor-use-cases

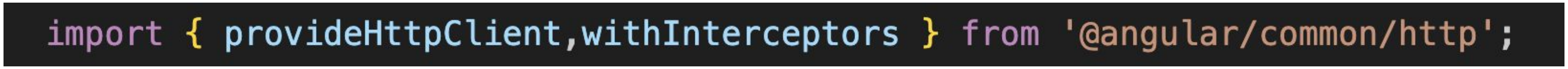
Interceptor (2)

Generate a new interceptor

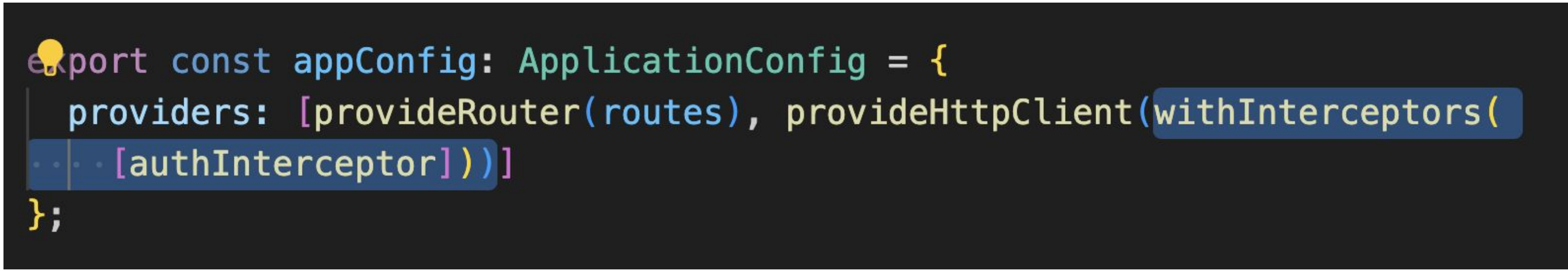
```
ng generate interceptor auth
```

Interceptor(3)

Add interceptor in app.config.ts



```
import { provideHttpClient,withInterceptors } from '@angular/common/http';
```



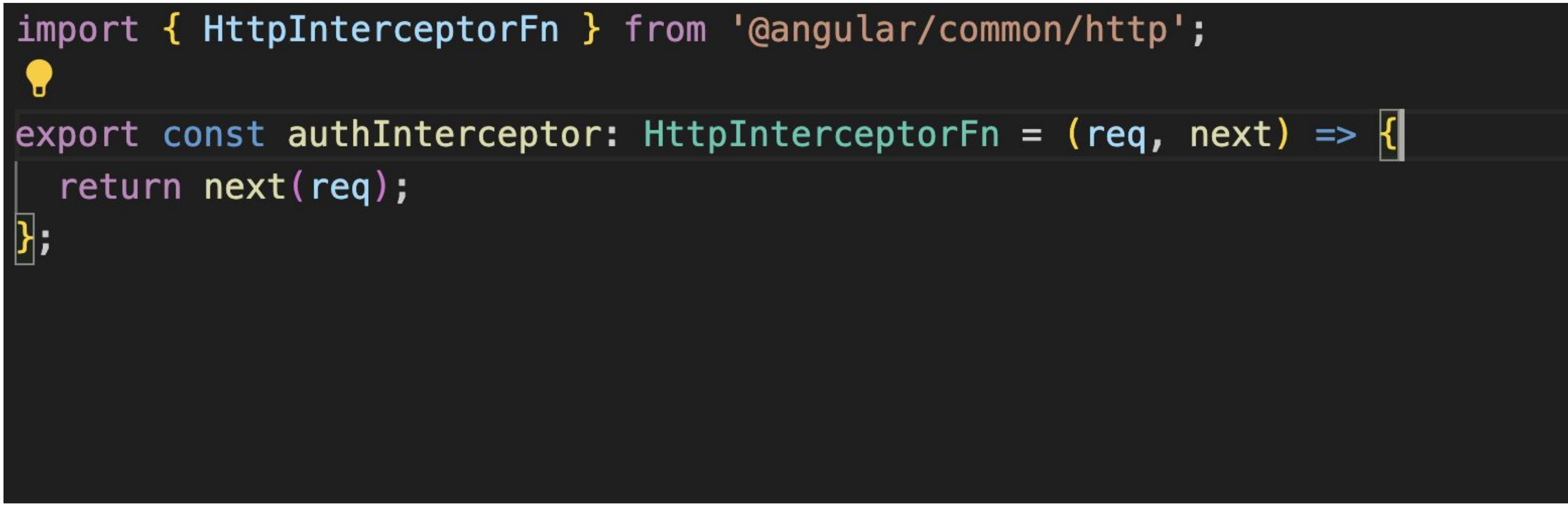
```
export const appConfig: ApplicationConfig = {
  providers: [provideRouter(routes), provideHttpClient(withInterceptors(
      [authInterceptor]))]
};
```

Interceptor (4)

The generated function `HttpInterceptorFn` accepts the following parameters:

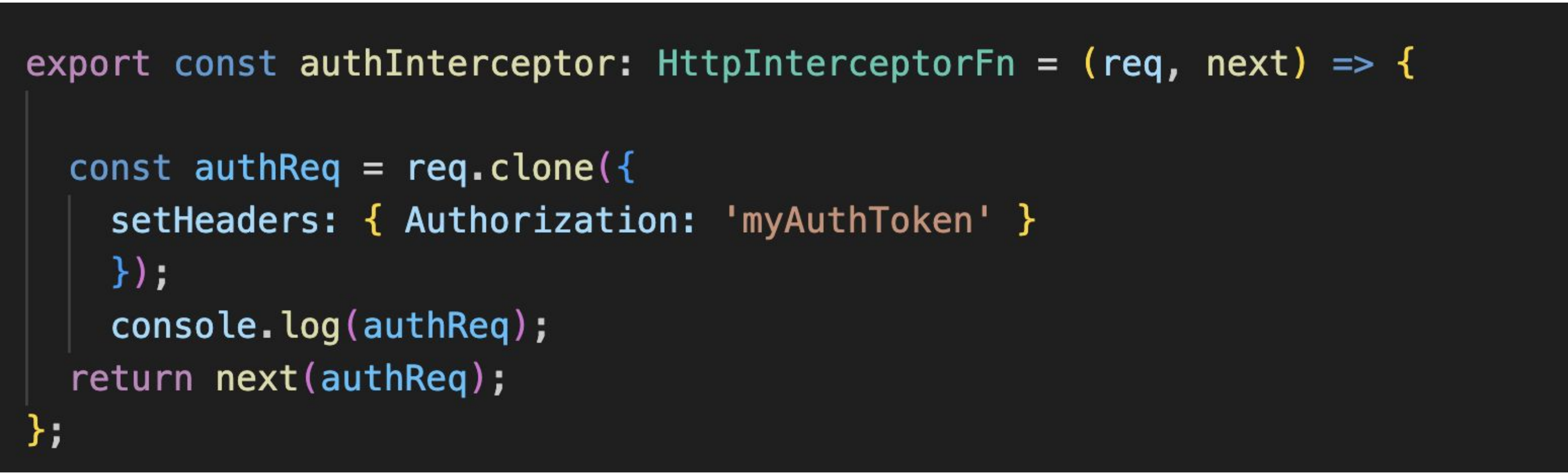
- req: An `HttpRequest` object that indicates the current request

• `next`: An `HttpHandler` object that denotes the next interceptor in the chain



```
import { HttpInterceptorFn } from '@angular/common/http';

export const authInterceptor: HttpInterceptorFn = (req, next) => {
  return next(req);
};
```

Interceptor (5)



```
export const authInterceptor: HttpInterceptorFn = (req, next) => {

  const authReq = req.clone({
    setHeaders: { Authorization: 'myAuthToken' }
    });
    console.log(authReq);
  return next(authReq);
};
```