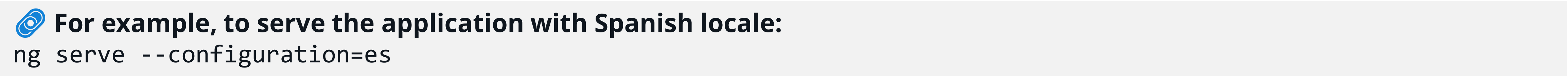
Angular 14 - 17

Angular and Internationalization

Angular and i18n

- Internationalization is a key aspect when creating multilanguage applications
- Angular provides us with a set of built-in features to deal with internationalization following the i18n standard:
  - Show dates, numbers, percentages in local formats.
  - Translate texts in component templates.
  - Handle plurals.
  - Handle alt text.
- By default angular uses the **locale** or regional settings in **en-US**.
- To set the locale to another locale we will use the **--i18n-locale** parameter to the CLI with the value we want to use (Angular follows the Unicode LDML convention.
  - https://unicode.org/reports/tr35/
  - E.g., we can also use the parameter --configuration



For example, to serve the application with Spanish locale:

```
ng serve --configuration=es
```

- These parameters can be used with **ng-serve** and with **ng-build**.

Angular and i18n

- We will need to insert the configuration entries in **angular.json**

```
"build": {
  ...
  "configurations": {
      ....
      "es": {
          "aot": true,
          "outputPath": "dist/myproject-es/",
          "i18nFile": "src/locale/messages.es.xlf",
          "i18nFormat": "xlf",
          "i18nLocale": "es",
      }
  }
}
...
"serve": {
  ...
  "configurations": {
      ....
      "es": {
"browserTarget": "TareasProyectos:build:es"
      }
  }
}
```

Pipes i18n

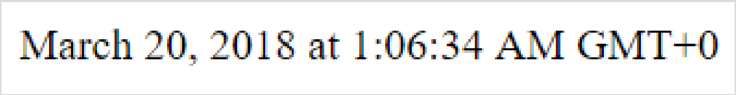
- Angular has a built-in set of pipelines to deal with some internationalization challenges like dates, decimals, and percentages.
- For example we can use a date pipe.
  - In our app.component.html, let's simply add a datePipe to display a dateTime property of our component with the 'long' property to the pipe to display the long version of a date.

```
<div>
   <span>{{dateTime|date:long}}</span>
</div>
```

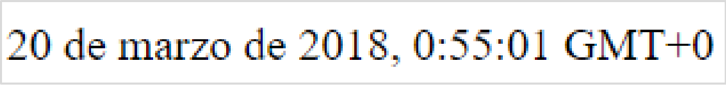
- In the component, just set the dateTime property to a new date.

```
 dateTime=new Date();
```

- If we run the application:
  - with "npm start", we will see the en-US version.
  - with “npm run start:es” the es-ES version.



```
March 20, 2018 at 1:06:34 AM GMT+0
```



```
20 de marzo de 2018, 0:55:01 GMT+0
```

Templates translation

- The translation of an angular i18n template has four phases:
    1. We will **mark** the static text for translation.
    2. The Angular i18n tool **extracts** the text from a standard translation source file.
    3. A translator (or ourselves) **edit** that file and return the text in the target language.
    4. The Angular **compiler** uses the translated file to generate a new version of the application with the target language.

Templates translation

Markup static text with angular i18n

- We are going to use the angular i18n attribute with a custom id, by doing so each translation unit generated by the i18n extraction tool is more readable and easier to maintain.
- We are also going to add a description and a meaning to the i18n attribute, this helps angular improve the accuracy of the translation.

```
<h1 i18n="[meaning]|[description]@@[id]"></h1>
```

- For example

```
<h1 i18n="site title|The title for the app@@appTtitle">The title</h1>
```

- The code above shows how to use these three values in the i18n attribute.
  - First, the meaning
  - then a pipe followed by the description.
  - Finally, with a double @ prefix for identification.

Templates translation

Using the i18n tool to extract the translation from a file

Let's run the next command to generate the main translation file:.

```
ng xi18n
```

- Our base translation file called "**messages.xlf**" will be generated in our src folder.
- Another location or name can be specified by passing them as parameters to the command:
  - **--output-path:** to indicate the path
  - **--out-file:** to indicate the name
  - **--i18n-locale:** to indicate the locale

- Now, we will create the new language-specific translation files.
  - We will create a "locale" folder in our src folder
  - We will copy the file "messages.xlf" in said folder and change the name to "messages.es.xlf".

Now, we will create the new language-specific translation files.

- Create a "**locale**" folder in our src folder
- Copy the file "**messages.xlf**" in that folder and change the name to "**messages.es.xlf**".

Templates translation

Edit the translation file

- In a real application, the company probably hires a translator who should be the one to provide the translated version of the files (the description and meaning will help this person to get a more accurate translation)
- To do the translation, we need to add a **<target>** tag after each of the **<source>** tags in the file.
- We'll write the translated version of the text in the target tag.

```
<trans-unit id="appTtitle" datatype="html">
    <source>Angular Internationalization</source>
    <target>Internacionalización de Angular</target>
    ...
    <note priority="1" from="description">The title for the app</note>
    <note priority="1" from="meaning">site title</note>
 </trans-unit>
```

Templates translation

Make the compilation in the specified language - AOT compiler

- The AOT (Ahead-of-Time) compiler is part of a build process that produces a small, fast, ready-to-run application package.
- When internationalizing with the AOT compiler, you must precompile a separate app bundle for each language and serve the appropriate bundle based on server-side language detection or url parameters.
- You also need to tell the AOT compiler to use your translation settings. To do so, you configure the translation with three options in your angular.json file.
    - **i18nFile**: the path to the translation file.
    - **i18nFormat**: the format of the translation file.
    - **i18nLocale**: the locale id.

Templates translation

Build in the specified language

- The **angular.json** file would look like this:

```
"build": {
  ...
  "configurations": {
    ...
    "fr": {
      "aot": true,
      "outputPath": "dist/my-project-fr/",
      "i18nFile": "src/locale/messages.es.xlf",
      "i18nFormat": "xlf",
      "i18nLocale": "es",
      ...
    }
  }
},
"serve": {
  ...
  "configurations": {
    ...
    "es": {
      "browserTarget": "*project-name*:build:es"
    }
  }
}
```

Templates translation

Make the compilation in the specified language - AOT compiler

- Then we pass the configuration with the command “ng serve” or “ng build”

```
ng serve --configuration=es
```

Templates translation

Build in the specified language

- The same configuration can also be executed from the CLI.
- We just need to add a couple of parameters to the Spanish build script in the package.json file.

```
"start":"ng serve"
"start:es":"ng serve --oat --i18nFile=src/locale/messages.es.xlf --i18nFormat=xlf --i18n-locale es"
```

  - **i18nFile:** points to the location of the translation file .
  - **i18nFormat:** refers to the format of the translation file.
  - **Locale:** specifies the locale

- To make build we will use a homonymous command

```
ng build --prod --i18n-file src/locale/messages.es.xlf --i18n-format xlf --i18n-locale es
```

Let's put it into practice: Tasks/Projects App

- Implement the English and Spanish version of the application.

Pipe-based dynamic translation

- Translations based on CLI features assumes building the application in the desired language.
- This approach has a limitation when implementing dynamic language changes.
- For cases in which a **dynamic implementation of language** change is needed, there is the possibility of implementing the translation using **Pipes**.
  - This option is the most natural in Angular, since the pipes already take into account the locale to make up the content output.

- To do this we will follow the following strategy
  1. Create **language files** in src/assets
  2. Create a **translation service**
  3. **Default language** loading
  4. Create the **translation pipes**
  5. Enable **dynamic language switching**

Pipe-based dynamic translation

Create language files in src/assets

- We will create as many translation files as languages we need.
- We will use the **json** format.
- The name of the file will be the equivalent of the locale of the language:

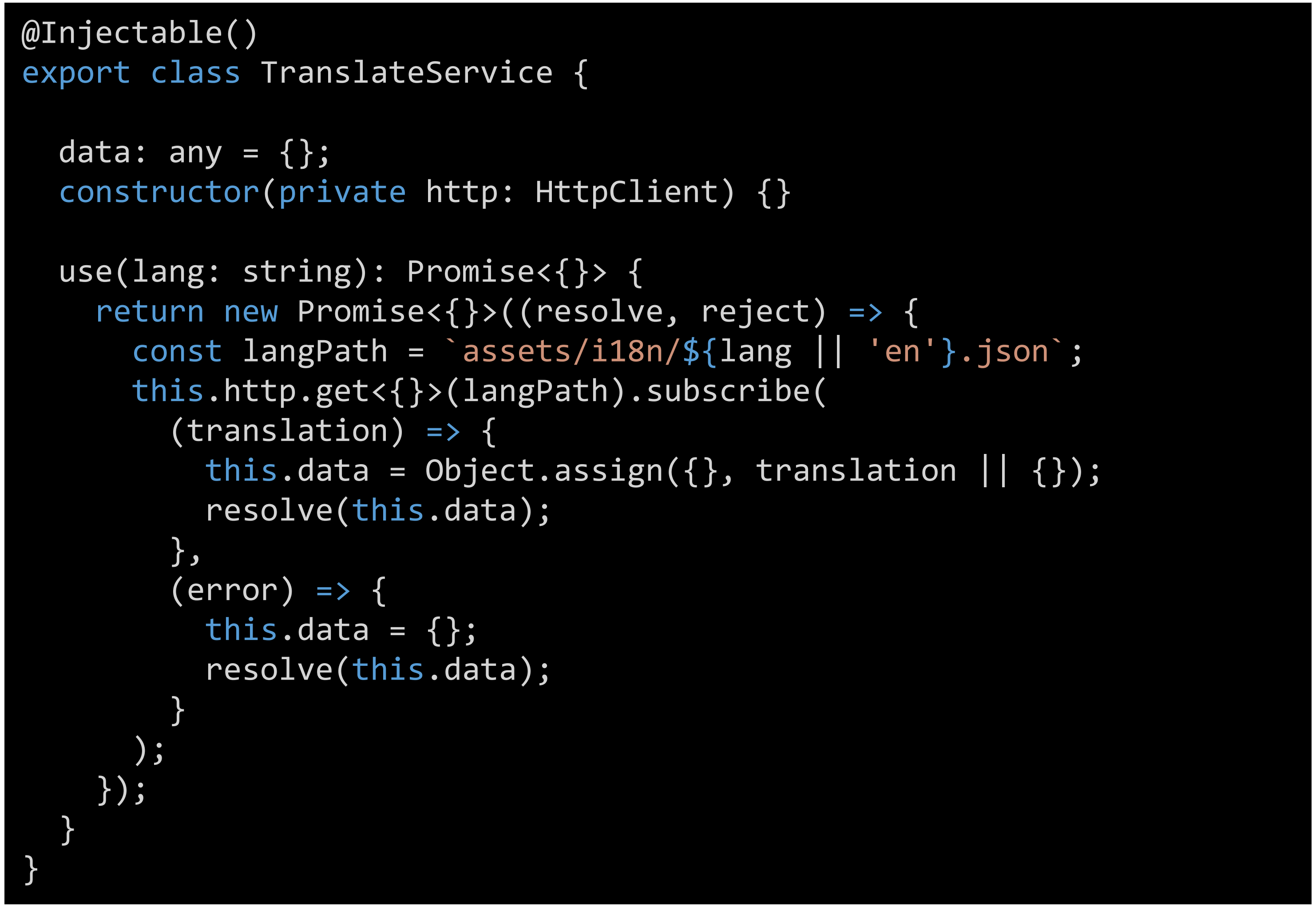
```
// file: en.json
{
  "TITLE": "My i18n Application (en)“
  ...
}
```

```
// file: es.json
{
  "TITLE": "Mi Aplicación i18n (es)“
  ...
}
```

Pipe-based dynamic translation

Create a translation service

- We will use a service that manages the translation of the entire application
- The service will load the corresponding language json file **based on the locale**.



```
@Injectable()
export class TranslateService {

  data: any = {};
  constructor(private http: HttpClient) {}

  use(lang: string): Promise<{}> {
    return new Promise<{}>((resolve, reject) => {
      const langPath = `assets/i18n/${lang || 'en'}.json`;
      this.http.get<{}>(langPath).subscribe(
        (translation) => {
          this.data = Object.assign({}, translation || {});
          resolve(this.data);
        },
        (error) => {
          this.data = {};
          resolve(this.data);
        }
      );
    });
  }
}
```

Pipe-based dynamic translation

Default language loading

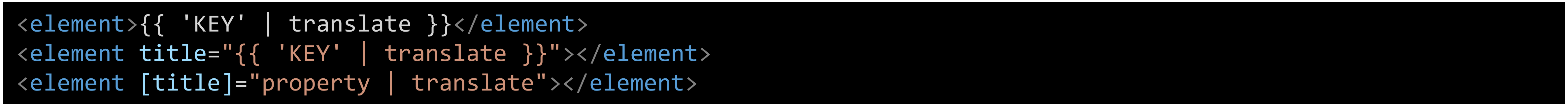
- We will define a default locale
- For this we will use the Angular **APP_INITIALIZER provider**, to which we will pass the value of the locale through a **setUp function:**

```
import { NgModule, APP_INITIALIZER } from '@angular/core';

export function setupTranslateFactory(
  service: TranslateService): Function {
  return () => service.use('en');
}

@NgModule({
  ...

  providers: [
    TranslateService,
    {
      provide: APP_INITIALIZER,
      useFactory: setupTranslateFactory,
      deps: [ TranslateService ],
      multi: true
    }
  ],

  ...
})
export class AppModule { }
```

Pipe-based dynamic translation

Create the translation pipes

- Using pipes is a common approach for text translation in Angular.
- We will use the following syntax for content translation in component template



```
<element>{{ 'KEY' | translate }}</element>
<element title="{{ 'KEY' | translate }}"></element>
<element [title]="property | translate"></element>
```

Pipe-based dynamic translation

Create the translation pipes

- Then, we will create a pipe that maps the value of the text according to the key
- We get the text using the translation service

```
import { Pipe, PipeTransform } from '@angular/core';
import { TranslateService } from './translate.service';
@Pipe({
  name: 'translate',
  pure: false,
})
export class TranslatePipe implements PipeTransform {
  constructor(private translate: TranslateService) {}
  transform(key: any): any {
    return this.translate.data[key] || key;
  }
}
```

- In our component we will use the pipe

```
<h1>
    {{ 'TITLE' | translate }}!
</h1>
```

Pipe-based dynamic translation

Enable dynamic language switching

- We will use an event mechanism (for example a click on a button) to change the language:

```
<div>
    <button (click)="setLang(es')">ES</button>
    <button (click)="setLang('en')">EN</button>
</div>
```

- In the event callback function we will call to the **use(lang)** method from the translation service

```
@Component({...})
export class AppComponent {
  constructor(private translate: TranslateService) {}

  ...

  setLang(lang: string) {
    this.translate.use(lang);
  }
}
```

Let's put it into practice: Tasks/Projects App

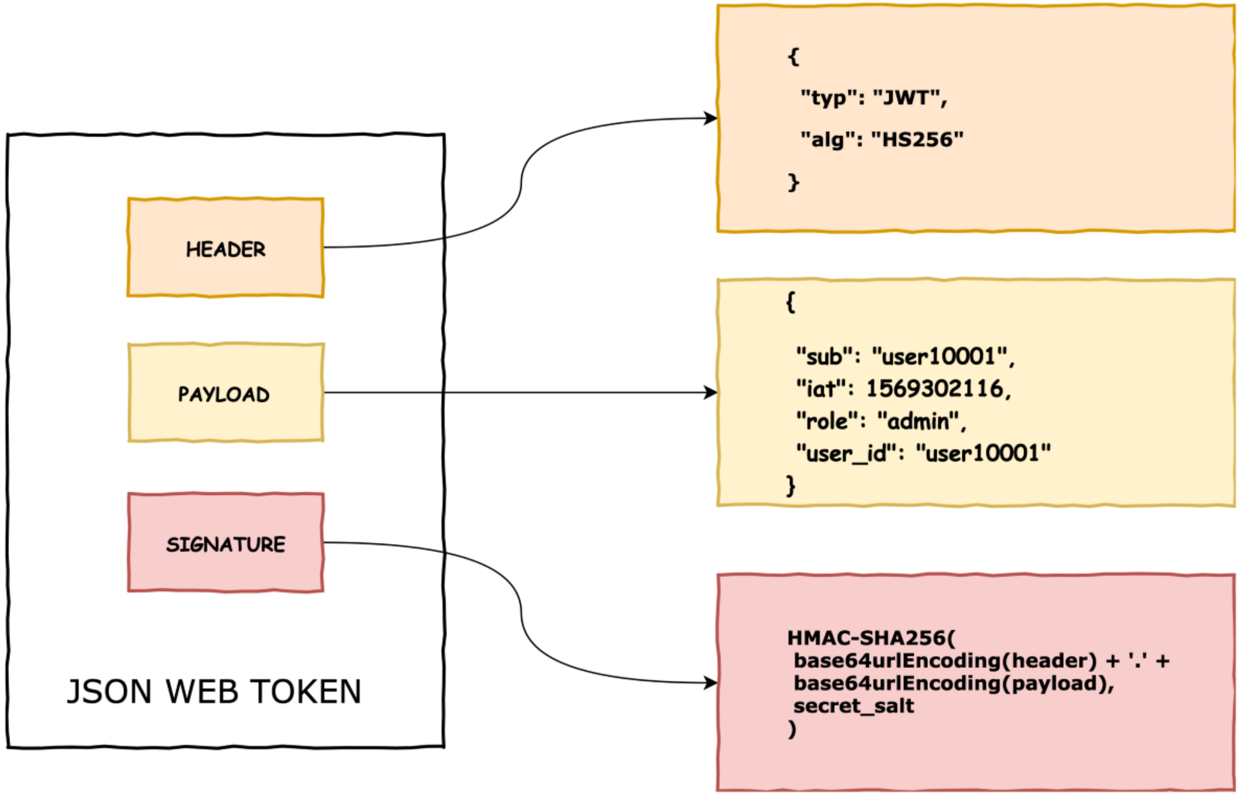
- Implement the dynamic version in English and Spanish of the application.

Authentication strategy

- Server-side web applications typically handle user sessions on the server.
  - They store the session details on the server and send the session ID to the browser via a cookie.
  - The browser stores the cookie and automatically sends it to the server with each request.
  - The server takes the session ID from the cookie and looks up the corresponding session details from its internal storage (memory, database, etc.).
  - Session details remain on the server and are not available on the client.

- In contrast, client-side web applications, such as Angular applications, manage user sessions on the client.
  - **Session data is stored on the client and sent to the server when needed.**
- A standardized way to store sessions on the client is **JSON Web Tokens**, also called **JWT** tokens .

JWT (JSON Web Token)



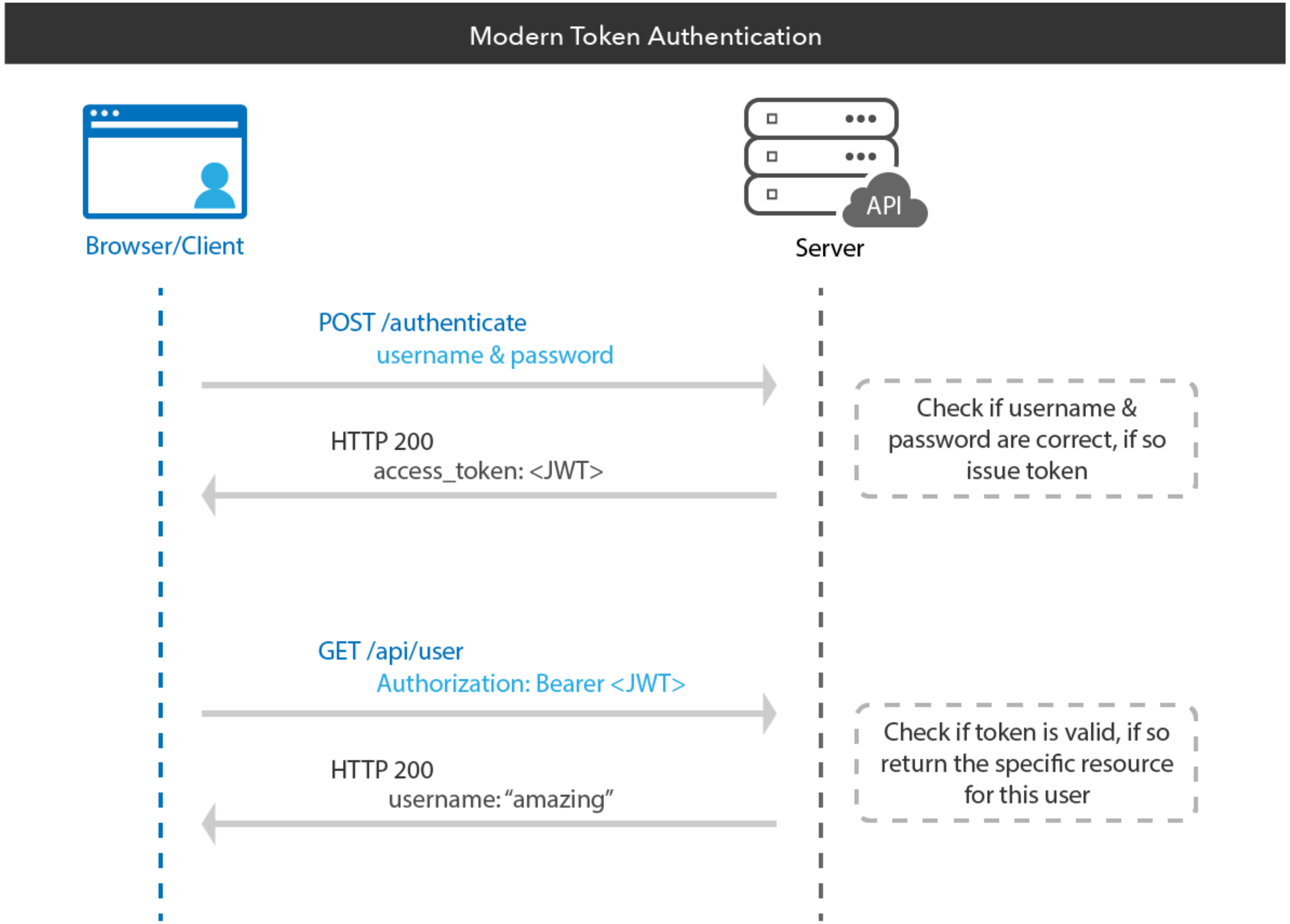
```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

```
{
 "sub": "user10001",
 "iat": 1569302116,
 "role": "admin",
 "user_id": "user10001"
}
```

```
HMAC-SHA256(
 base64urlEncoding(header) + '.' +
 base64urlEncoding(payload),
 secret_salt
)
```

- JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact, self-contained way to securely transmit information between parties as a JSON object.
- This information can be verified and trusted because it is digitally signed.
- JWTs can be signed using a secret (using the HMAC algorithm) or a public/private key pair using RSA or ECDSA.
- https://jwt.io/

JWT (JSON Web Token)



Modern Token Authentication

Browser/Client

Server

POST /authenticate
username & password

Check if username & password are correct, if so issue token

HTTP 200
access_token: <JWT>

GET /api/user
Authorization: Bearer <JWT>

Check if token is valid, if so return the specific resource for this user

HTTP 200
username: "amazing"

Dissecting the JWT token

Token header
is a Base64URL encoded JSON object. Contains information that describes the type of token and the signature algorithm being used, such as HMAC, SHA256, or RSA.

```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

Payload
contains something called claims, which are statements about the entity (usually the user) and additional data. There are three different types of claims: registered, public and private. The claims are the most "interesting" part of a JWT token, since they contain data about the user in question.

```
{
 "ver": 1,
 "jti": "AT.u_OOxGzWwTcDYlxfpp5X_3quR0vRnsnXmwLfWtL1cto",
 "iss": "https://dev-819633.oktapreview.com/oauth2/default",
 "aud": "api://default",
 "iat": 1546726228,
 "exp": 1546729974,
 "cid": "0oaiox8bmsBKVXku30h7",
 "scp": [
  "customScope"
 ],
 "sub": "0oaiox8bmsBKVXku30h7"
}
```

Signature
it is created by taking the encrypted header, the encrypted payload, a secret key, and using the algorithm specified in the header to cryptographically sign these values.

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret
)
```

Benefits

- JWT is a stateless authentication mechanism as the user state is never saved in the database.
- JWTs are self-contained: all the necessary information is there, reducing the need of going back and forward to the database. With JWT we don't need to query database to authenticate the user for every api call.
- Protects against CSRF (Cross Site Request Forgery) attacks.
- JWT is compact. Because of its size, it can be sent through an URL, POST parameter, or inside an HTTP header.
- You can authorize only the requests you wish to authorize. Cookies are sent for every single request.
- You can send JWT to any domain. This is especially useful for single page applications that are consuming multiple services that are requiring authorization - so I can have a web app on the domain myapp.com that can make authorized client-side requests to myservice1.com and to myservice2.com. Cookies are bound to a single domain. A cookie created on the domain foo.com can't be read by the domain bar.com.

Session service

- Now that we have an API method to authenticate with our back-end, we need a mechanism to store the token we receive from the API.
- Because the data will be unique across our entire application, we'll store it in a service called **SessionService**.
- So let's generate our new SessionService:

```
@Injectable({
  providedIn: 'root',
})
export class SessionService {
  public token: string = '';
  public name: string = '';

  constructor() {}

  public destroy(): void {
    this.token = '';
    this.name = '';
  }
}
```

Session service

- We define a property to store the user's API access token and name.
- We also add a **destroy()** method to reset all data in case we want to log out the current user.
- The SessionService is not aware of any authentication logic. It is only responsible for storing session data.
- We'll create a separate AuthService to implement the actual authentication logic.

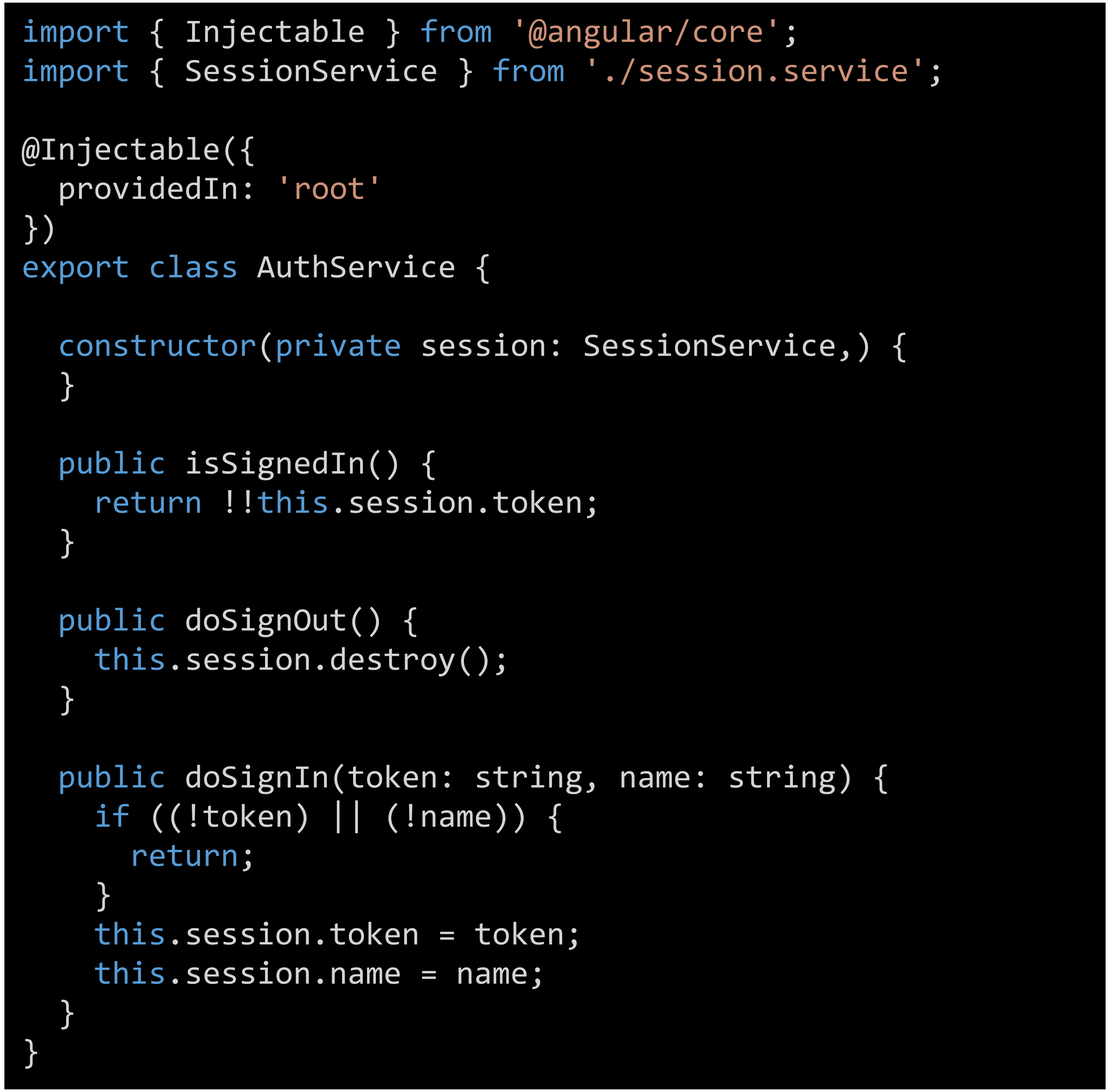
Login in the API service

- We will use an Angular service to take care of accessing the API we will add a method that sends the data for authentication

```
@Injectable()
export class ApiService {

  constructor(private http: HttpClient) {
  }

  public signIn(username: string, password: string): Observable<any> {
    return this.http.post(`${API_URL}/sign-in`, {
      username,
      password
    }).pipe(catchError(this.errorHandl));
  }


  errorHandl = (error: any) => {...};
}
```

Authentication service

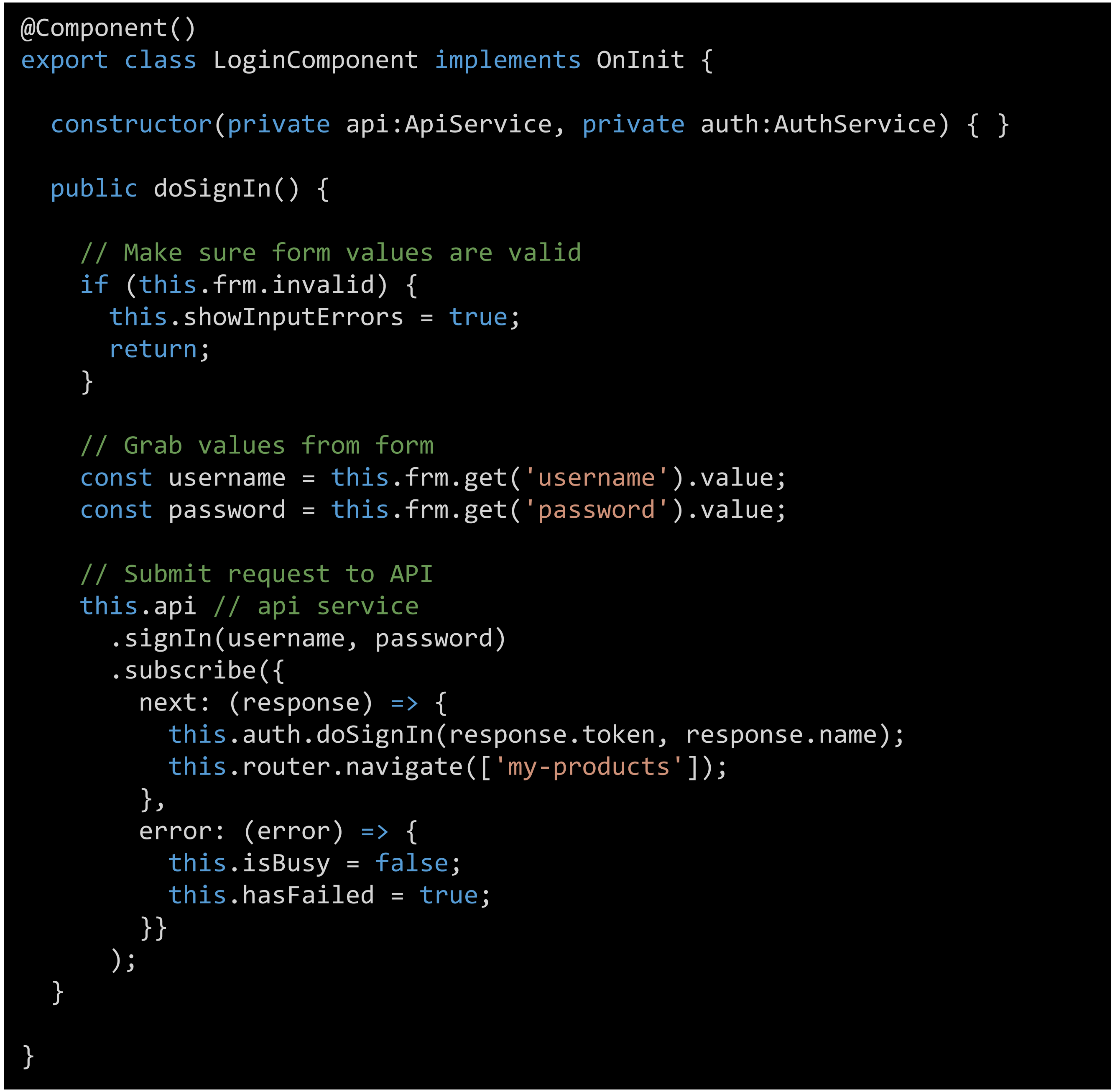
- By putting the authentication logic in a separate service, the separation between the authentication process and the storage of session data is promoted.
- This ensures that we don't have to change the SessionService if the authentication flow changes and allows us to easily mock session data in unit tests.
- We have injected SessionService and added some methods:
    - **isSignedIn():** returns whether the user is signed in or not
    - **doSignOut():** Signs out the user by clearing the session data
    - **doSignIn():** registers the user by storing the session data.



```
import { Injectable } from '@angular/core';
import { SessionService } from './session.service';

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

  constructor(private session: SessionService,) {
  }

  public isSignedIn() {
    return !!this.session.token;
  }

  public doSignOut() {
    this.session.destroy();
  }

  public doSignIn(token: string, name: string) {
    if ((!token) || (!name)) {
      return;
    }
    this.session.token = token;
    this.session.name = name;
  }
}
```

Login component

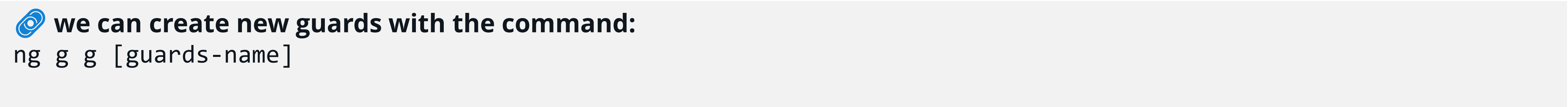
- A component with a form to enter a username and password
- These data, if valid, will be sent to the authentication service, which in turn will store the user and the token
- Finally we will redirect to the private entry page
- We will implement all this in a **doSignIn** method



```
@Component()
export class LoginComponent implements OnInit {

  constructor(private api:ApiService, private auth:AuthService) { }

  public doSignIn() {

    // Make sure form values are valid
    if (this.frm.invalid) {
      this.showInputErrors = true;
      return;
    }

    // Grab values from form
    const username = this.frm.get('username').value;
    const password = this.frm.get('password').value;

    // Submit request to API
    this.api // api service
      .signIn(username, password)
      .subscribe({
        next: (response) => {
          this.auth.doSignIn(response.token, response.name);
          this.router.navigate(['my-products']);
        },
        error: (error) => {
          this.isBusy = false;
          this.hasFailed = true;
        }}
      );
  }

}
```

Route Guards: protecting our private routes

- In essence, a route guard is a **function that returns true** to indicate that routing is allowed or **false** to indicate that routing is not allowed.
- A guard can also return a **Promise** or **Observable** that evaluates to true or false.
- In that case, the router will wait until the Promise or Observable completes.
- There are 4 types of route guards:
  - **CanLoad:** determines whether a module can be lazy-loaded (lazy-loaded module)
  - **CanActivate:** Determines whether a path can be activated when the user navigates to that path.
  - **CanActivateChild:** Determines whether a route can be activated when the user navigates to one of its children.
  - **CanDeactivate:** Determines if a route can be deactivated.
- In our app, we want to make sure that the user is logged in when browsing ‘my-products’ route.
  - Therefore, a **CanActivate** protector is a good option.

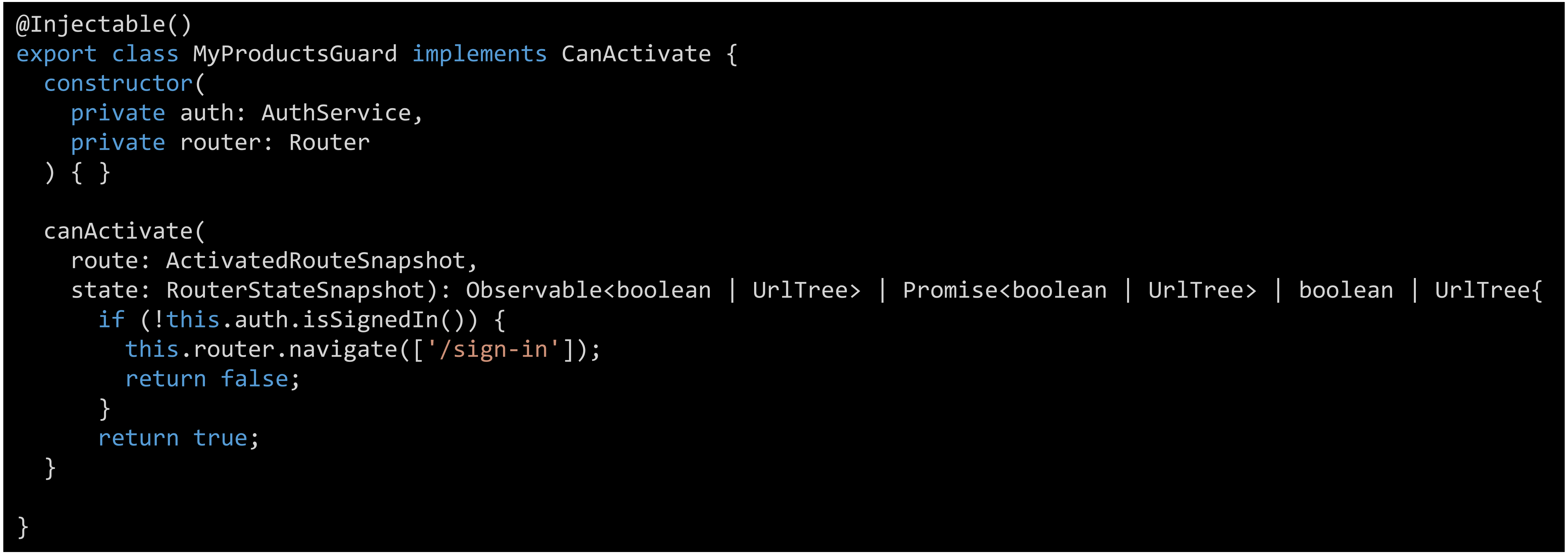


we can create new guards with the command:

```
ng g g [guards-name]
```

Route Guards: protecting our private routes

- In our app, we want to make sure that the user is logged in when browsing everyone's route. Therefore, CanActivate is a good option.
- We will create a service that implements the function



```
@Injectable()
export class MyProductsGuard implements CanActivate {
  constructor(
    private auth: AuthService,
    private router: Router
  ) { }

  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree{
      if (!this.auth.isSignedIn()) {
        this.router.navigate(['/sign-in']);
        return false;
      }
      return true;
  }

}
```

Route Guards: protecting private routes

- The **canActivate**() method receives the activated route and the router state as arguments , in case we need to make a decision whether or not to allow navigation.
  - In our example, the logic is very simple. If the user is not logged in, we ask the angular router to navigate the user to the login page and stop further navigation.
  - Conversely, if the user is logged in, true is returned, allowing the user to navigate to the requested path.

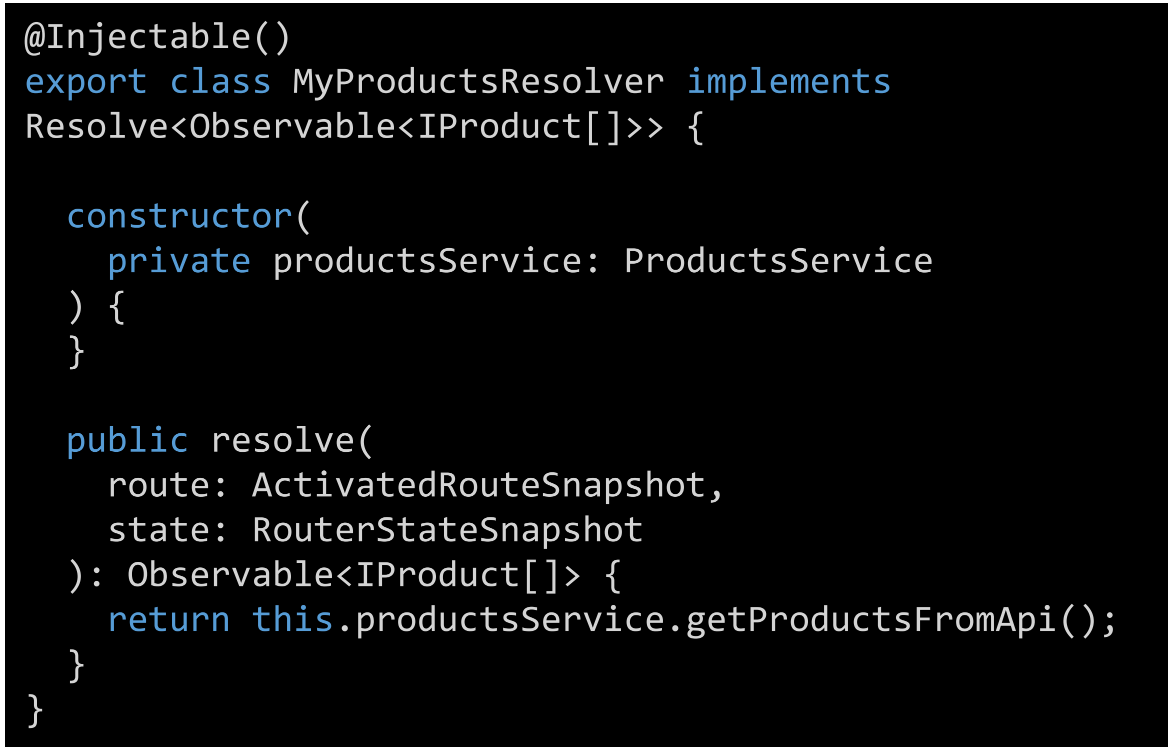
Route Guards: protecting private routes

- Once the route guard is created, we must tell the Angular router to use it.
- To do this we add **canActivate** to the desired route
- canActivate accepts an **array of guards** of type CanActivate.

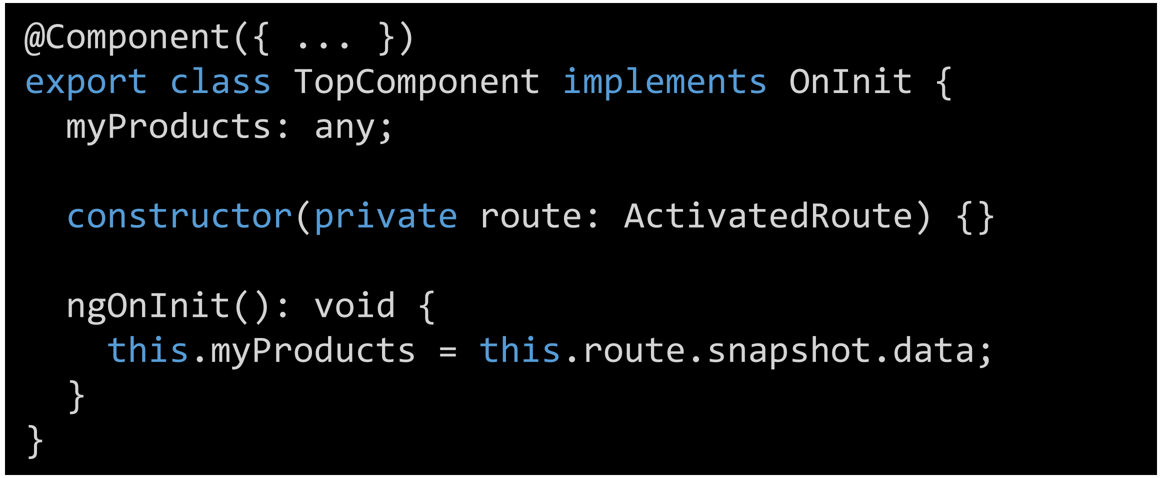
```
...
const routes: Routes = [
  {
    path: '',
    redirectTo: 'sign-in',
    pathMatch: 'full'
  },
  {
    path: 'sign-in',
    component: SignInComponent
  },
  {
    path: 'my-products',
    component: MyProductsComponent,
    canActivate: [
      MyProductsGuard
    ],
    resolve: {
      products: ProductsResolver
    }
  },
  {
    path: '**',
    component: PageNotFoundComponent
  }
];
...
```

Resolvers

- In some circumstances, while the app is getting the data from the API, the component can show a loading indicator or similar.
- There is another way to use what is known as a **route resolver**, which allows you to get data before navigating to the new route.
- A resolver is a class with a method that acts as a **data provider** for the page's initialization and makes the router wait for the data to be resolved before the route is finally activated.
- In the component, we can access the resolved data using the data property of ActivatedRoute's snapshot object:



```
@Injectable()
export class MyProductsResolver implements
Resolve<Observable<IProduct[]>> {

  constructor(
    private productsService: ProductsService
  ) {
  }

  public resolve(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Observable<IProduct[]> {
    return this.productsService.getProductsFromApi();
  }
}
```



```
@Component({ ... })
export class TopComponent implements OnInit {
  myProducts: any;

  constructor(private route: ActivatedRoute) {}

  ngOnInit(): void {
    this.myProducts = this.route.snapshot.data;
  }
}
```

Sending a token with the request

- In our service, we define a method to create the request options

```
private getRequestOptions() {
  const headers = new Headers({
    'Authorization': 'Bearer ' + this.session.token
  });
  return new RequestOptions({ headers });
}
```

- We update the methods that communicate with the API

```
public getMyProducts(): Observable<IProduct[]> {
  const options = this.getRequestOptions();
  return this.http
    .get(`${API_URL}/my-products`, options);
    ...
}
```

Let's put it into practice: Tasks/Projects App

- It implements an authentication architecture with route guards for the application.

Next steps

We would like to know your opinion!

Please, let us know what you think about the content.
From Netmind we want to say thank you, we appreciate time
and effort you have taking in answering all of that is
important in order to improve our training plans so that you
will always be satisfied with having chosen us
quality@netmind.es

[ııı] netmind
a bts company

Thanks!

Follow us: