

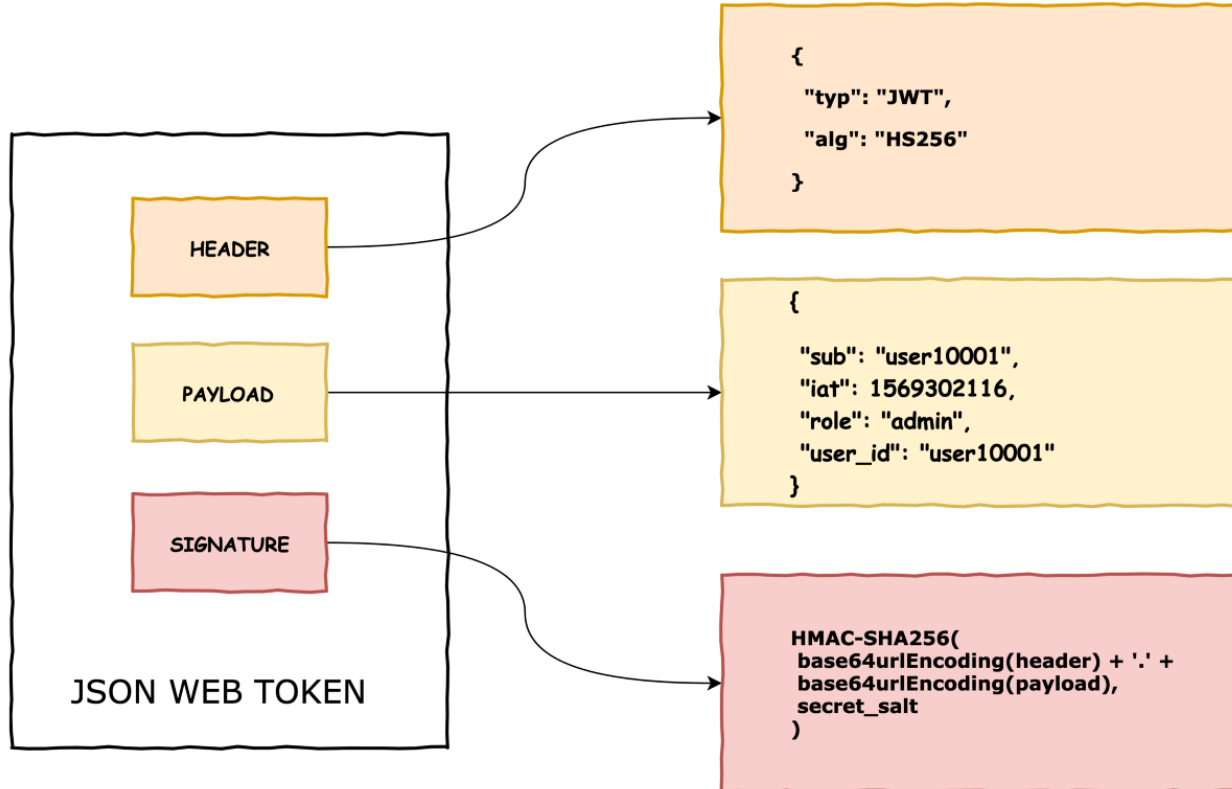
Angular 14 - 16

# Authentication and route protection

# Authentication strategy

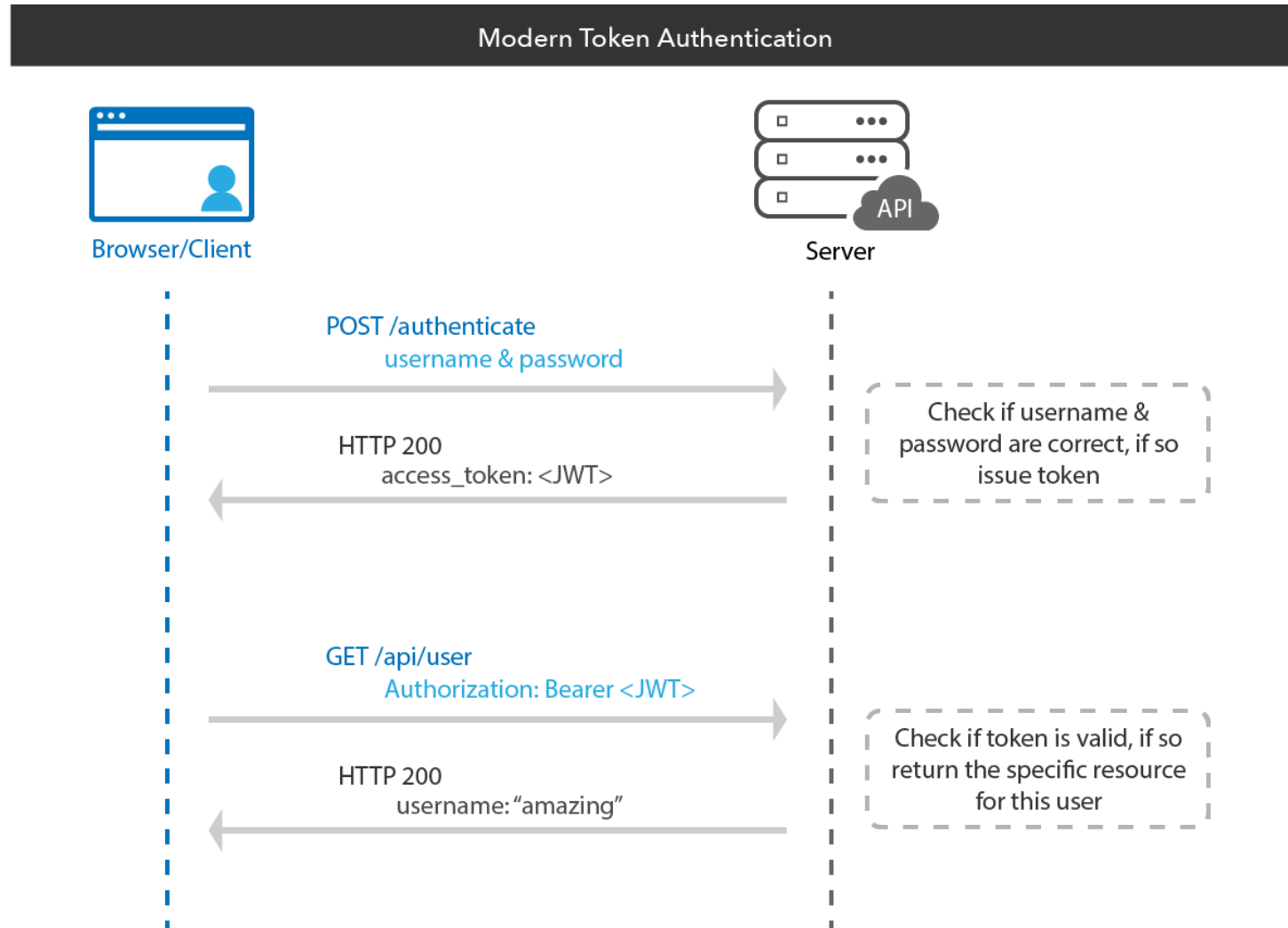
- Server-side web applications typically handle user sessions on the server.
  - They store the session details on the server and send the session ID to the browser via a cookie.
  - The browser stores the cookie and automatically sends it to the server with each request.
  - The server takes the session 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)



- 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)



# 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_00xGzWwTcDY1xfpp5X_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.handleError));
  }

  handleError = (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 productService: ProductService
    ) {

    }

    public resolve(
        route: ActivatedRouteSnapshot,
        state: RouterStateSnapshot
    ): Observable<IProduct[]> {
        return this.productService.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](mailto:quality@netmind.es)

# Thanks!

Follow us:

