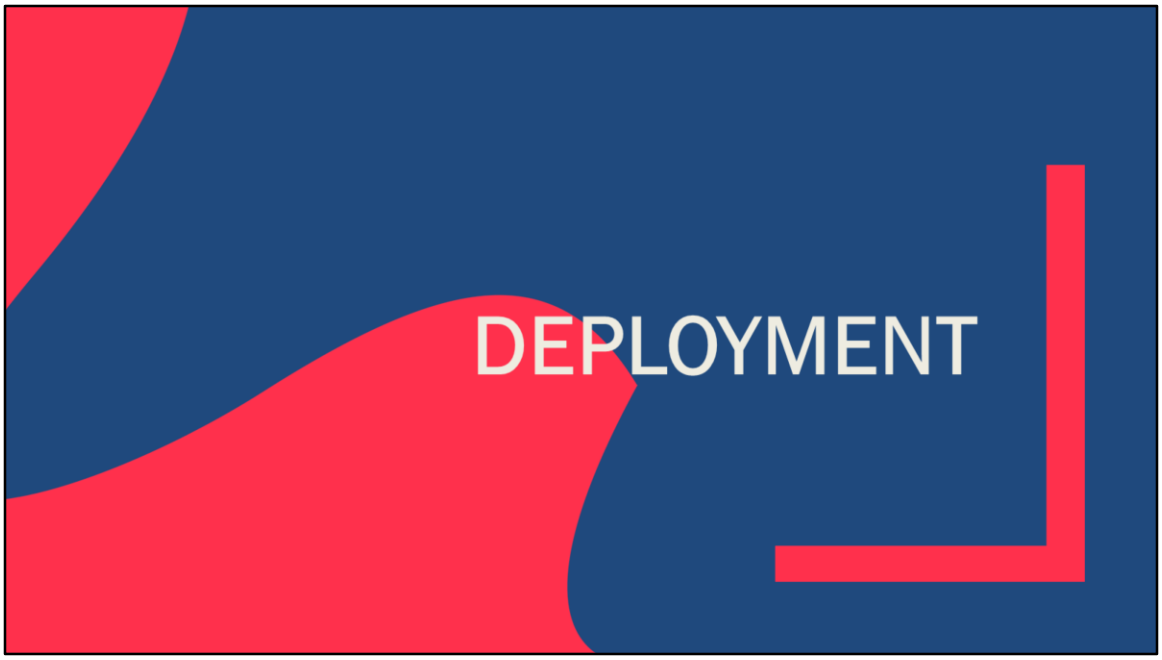

ANGULAR 6
Lesson 09


DEPLOYMENT /
AUTHENTICATION
Objectives

Capgemini

Deployment / Authentication

- Deployment Preparations
- JIT vs AOT Compilation
- Setup development env and backend env
- Environments configuration
- Linting
- Deployment
- How Authentication works in SPA
- JSON Web Tokens
- Signup, Login and logout application
- Router Protection, Route Guards
- CanActivate interface
- Checking and using Authentication Status


DEPLOYMENT

Capgemini

Deployment Preparations

- When you are ready to deploy your Angular application to a remote server, you have various options for deployment.
- SIMPLEST DEPLOYMENT
  - *For the simplest deployment, create a production build and copy the output directory to a web server.*
  - *Start with the production build:*
    - > ng build --prod
  - *Copy everything within the output folder (dist/ by default) to a folder on the server.*
  - *Configure the server to redirect requests for missing files to index.html.*

  This is the simplest production-ready deployment of your application

Capgemini

WHAT ARE ROUTE GUARDS?

- Angular's route guards are interfaces which can tell the router whether or not it should allow navigation to a requested route. They make this decision by looking for a true or false return value from a class which implements the given guard interface.

- There are five different types of guards and each of them is called in a particular sequence. The router's behavior is modified differently depending on which guard is used.

The guards are:

- *CanActivate*
- *CanActivateChild*
- *CanDeactivate*
- *CanLoad*
- *Resolve*

Capgemini

JIT vs AOT Compilation

Capgemini

Setup development env and backend env

Capgemini

Environments configuration

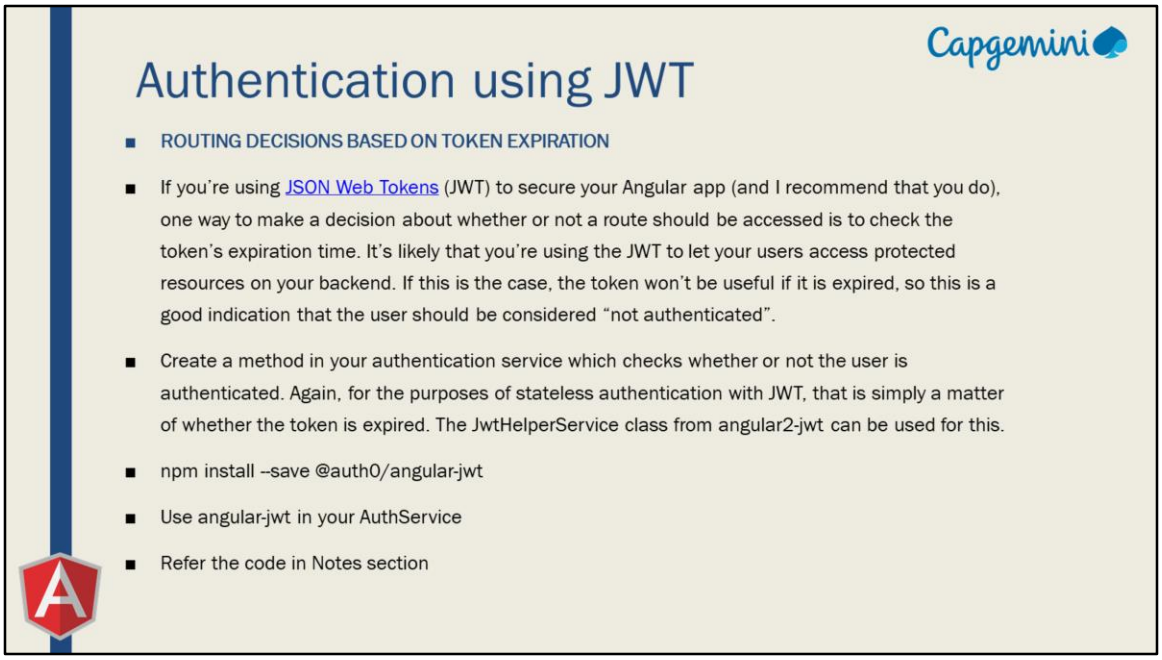
Capgemini

Linting

Capgemini

Deployment


AUTHENTICATION




```
// src/app/auth/auth.service.ts
import { Injectable } from '@angular/core';
import { JwtHelperService } from '@auth0/angular-jwt';
@Injectable()
export class AuthService {
  constructor(public jwtHelper: JwtHelperService) {}
  // ...
  public isAuthenticated(): boolean {
    const token = localStorage.getItem('token');
    // Check whether the token is expired and return
    // true or false
    return !this.jwtHelper.isTokenExpired(token);
  }
}
```

CHECKING FOR A USER'S ROLE

- THE ABOVE EXAMPLE WORKS WELL FOR SCENARIOS THAT ARE FAIRLY STRAIGHT FORWARD. IF A USER IS AUTHENTICATED, LET THEM PASS. THERE ARE MANY CASES, HOWEVER, WHERE WE'LL WANT TO BE A BIT MORE FINE-GRAINED WITH OUR ROUTING DECISIONS.
- FOR EXAMPLE, WE MAY WISH TO ONLY PERMIT ACCESS TO A ROUTE FOR USERS THAT HAVE A CERTAIN ROLE ATTACHED TO THEIR ACCOUNT. TO HANDLE THESE CASES WE CAN MODIFY THE GUARD TO LOOK FOR A CERTAIN ROLE IN THE PAYLOAD OF THE USER'S JWT.
- INSTALL JWT-DECODE SO WE CAN READ THE JWT PAYLOAD.
- NPM INSTALL --SAVE JWT-DECODE
- SINCE THERE WILL BE TIMES THAT WE WANT TO USE BOTH THE CATCH-ALL AUTHGUARDAND A MORE FINE-GRAINED ROLE-BASED GUARD, LET'S CREATE A NEW SERVICE SO WE CAN HANDLE BOTH CASES.

Create a new guard service called RoleGuardService.

```
// src/app/auth/role-guard.service.ts
import { Injectable } from '@angular/core';
import {
  Router,
  CanActivate,
  ActivatedRouteSnapshot
} from '@angular/router';
import { AuthService } from './auth.service';
import decode from 'jwt-decode';
@Injectable()
export class RoleGuardService implements CanActivate {
  constructor(public auth: AuthService, public router: Router) {}
  canActivate(route: ActivatedRouteSnapshot): boolean {
    // this will be passed from the route config
    // on the data property
    const expectedRole = route.data.expectedRole;
    const token = localStorage.getItem('token');
    // decode the token to get its payload
```

```
    const tokenPayload = decode(token);
    if (
      !this.auth.isAuthenticated() ||
      tokenPayload.role !== expectedRole
    ) {
      this.router.navigate(['login']);
      return false;
    }
    return true;
  }
}
```

In this guard we're using ActivatedRouteSnapshot to give us access to the data property for a given route. This data property is useful because we can pass an object with some custom properties to it from our route configuration. We can then pick up that custom data in the guard to help with making routing decisions. In this case we're looking for a role that we expect the user to have if they are to be allowed access to the route. Next we are decoding the token to grab its payload. If the user isn't authenticated **or** if they don't have the role we expect them to have in their token payload, we cancel navigation and have them log in. Otherwise, they are free to proceed.

We can now use this RoleGuardService for any of our routes. We might, for example, want to protect an /admin route.

```
// src/app/app.routes.ts
import { Routes, CanActivate } from '@angular/router';
import { ProfileComponent } from './profile/profile.component';
import {
  AuthGuardService as AuthGuard
} from './auth/auth-guard.service';
import {
  RoleGuardService as RoleGuard
```

```
} from './auth/role-guard.service';
export const ROUTES: Routes = [
  { path: '', component: HomeComponent },
  {
    path: 'profile',
    component: ProfileComponent,
    canActivate: [AuthGuard]
  },
  {
    path: 'admin',
    component: AdminComponent,
    canActivate: [RoleGuard],
    data: {
      expectedRole: 'admin'
    }
  },
  { path: '**', redirectTo: '' }
];
```

For the /admin route, we're still using canActivate to control navigation, but this time we're passing an object on the data property which has that expectedRole key that we've already seen in the RoleGuardService.
Note: This scenario assumes that you are using a custom role claim in your JWT.

Capgemini

How Authentication works in SPA

Capgemini

JSON Web Tokens

Capgemini

Signup, Login and logout application

Capgemini

Router Protection, Route Guards

Capgemini

CanActivate interface

Capgemini

Checking and using Authentication Status