JWT Authentication

1 JWT

Definition in https://jwt.io/introduction/:
„JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed.“

Definition auf https://de.wikipedia.org/wiki/JSON_Web_Token:
Ein **JSON Web Token** (**JWT**, vorgeschlagene Aussprache: [dʒɒt]) ist ein auf JSON basiertes und nach RFC 7519 genormtes Access-Token. Das JWT ermöglicht den Austausch von verifizierbaren Claims. Es wird typischerweise verwendet, um in einem System mit einem Drittanbieter die Identität eines Benutzers zwischen einem Identity-Provider und einem Service-Provider auszutauschen. Des Weiteren eignet sich JWT zur Implementierung einer Stateless Session, denn da alle für die Authentifikation benötigten Informationen in dem Token übertragen werden, muss die Sitzung nicht auf dem Server gespeichert werden.

Man erstellt ein Projekt wie gewohnt, dabei ist es prinzipiell egal ob mit oder ohne Routing. Um aber auch am Frontend Seiten zu schützen, müssen Routen definiert werden.

2 Ausgangssituation

Es soll mit einem sehr einfachen Projekt begonnen werden.

2.1 Backend

Es gibt vorerst nur einen ValuesController der eine GET-Action bereitstellt

```
[Route("[controller]/[action]")]
[ApiController]
public class ValuesController : ControllerBase
{
  [HttpGet]
  public DummyDataDto Dummy()
  {
    Console.WriteLine($"ValuesController::DummyData");
    return new DummyDataDto
    {
      IntVal = DateTime.Now.Second,
      StringVal = $"{DateTime.Now:HH:mm:ss}",
    };
  }
}
```

```
public class DummyDataDto
{
  public int IntVal { get; set; }
  public string StringVal { get; set; }
}
```

2.2 Frontend

Das Frontend liest diese Daten über ein Service und zeigt das Ergebnis an. Der Einfachheit halber wird das in einer Komponte HomeComponent programmiert, die zwar über Routing erreichbar, aber nicht in einem eigenen Feature-Modul liegt.

```
export class HomeComponent implements OnInit {
  dummyData: Observable<DummyDataDto>;

  constructor(private valuesService: ValuesService) { }

  ngOnInit(): void {
    console.log('HomeComponent::ngOnInit()');
    this.dummyData = this.valuesService.getDummyData();
  }

}
```

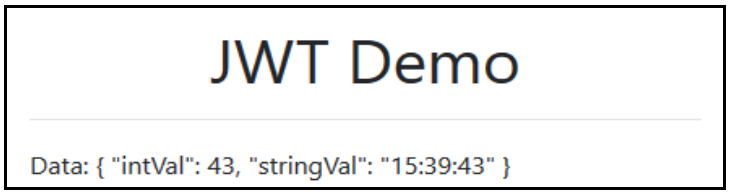
```
<div>Data: {{dummyData | async | json}}</div>
```

```
export class ValuesService {

  private urlBase = 'http://localhost:5000/values';
  constructor(private httpClient: HttpClient) { }

  getDummyData(): Observable<DummyDataDto> {
    return this.httpClient.get<DummyDataDto>(`${this.urlBase}/dummy`);
  }
}
```

Und so sieht es aus:



2.2.1 Routing

Beim Routing gibt es keine Überraschungen:

```
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: '**', redirectTo: '' }// otherwise redirect to home
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

2.3 Authorize

Jetzt kann man einen Controller sehr einfach schützen, indem man **`[Authorize]`** über die Klassendeklaration bzw. über eine Methode schreibt. Dabei gilt:

- Über der Klasse ➔ alle Methoden sind geschützt
- Über der Methode ➔ diese Methode ist geschützt

Im ersten Fall kann man den Schutz durch **`[AllowAnonymous]`** aufheben.

```
[Authorize]
[Route("[controller]/[action]")]
[ApiController]
public class ValuesController : ControllerBase
{
  [HttpGet]
  public DummyDataDto Dummy()
  {
    Console.WriteLine($"ValuesController::Dummy");
    return new DummyDataDto
    {
      IntVal = DateTime.Now.Second,
      StringVal = $"{DateTime.Now:HH:mm:ss}",
    };
  }

  [AllowAnonymous]
  [HttpGet]
  public DummyDataDto DummyUnchecked()
  {
    Console.WriteLine($"ValuesController::DummyUnchecked");
    return new DummyDataDto
    {
      IntVal = DateTime.Now.Second,
      StringVal = $"{DateTime.Now:HH:mm:ss}",
    };
  }

}
```

In diesem Beispiel ist Dummy() geschützt (weil durch Authorize über der Klasse prinzipiell alle Methoden geschützt sind), DummyUnchecked() kann aber aufgeruren werden, weil das durch AllowAnonymous expizit erlaubt wurde.

2.3.1 Frontend

Jetzt erhält man von der geschützten Action keine Daten mehr, von der ungeschützten jedoch schon:

```
export class HomeComponent implements OnInit {
  dummyData: Observable<DummyDataDto>;
  dummyDataUnchecked: Observable<DummyDataDto>;

  constructor(private valuesService: ValuesService) { }

  ngOnInit(): void {
    console.log('HomeComponent::ngOnInit()');
    this.dummyData = this.valuesService.getDummyData();
    this.dummyDataUnchecked = this.valuesService.getDummyDataUnchecked();
  }
}
```

```
<div>Checked: {{dummyData | async | json}}</div>
<div>Unchecked: {{dummyDataUnchecked | async | json}}</div>
```

```
Checked: null
Unchecked: { "intVal": 48, "stringVal": "09:34:48" }
```

In der Browser-Kosnole sieht man eine entsprechende Fehlermeldung (Fehlercode 401):

```
HomeComponent::ngOnInit()                                   home.component.ts:19:12
ERROR                                                                  core.js:4197
  Object { headers: {…}, status: 401, statusText: "Unauthorized", url:
  "http://localhost:5000/values/dummy", ok: false, name: "HttpErrorResponse",
  message: "Http failure response for http://localhost:5000/values/dummy: 401
  Unauthorized", error: null }
```

Am Backend wird eine entsprechende Exception geworfen:

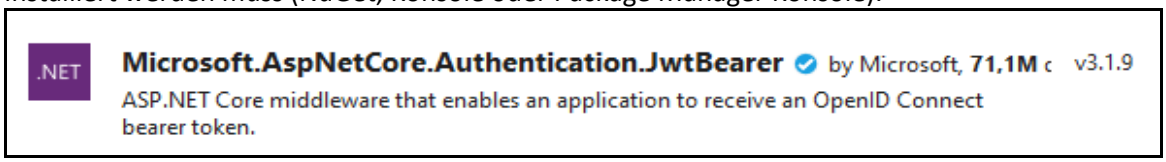
```
fail: Microsoft.AspNetCore.Diagnostics.DeveloperExceptionPageMiddleware[1]
      An unhandled exception has occurred while executing the request.
System.InvalidOperationException: No authenticationScheme was specified, and there was no DefaultChallengeScheme found.
```

3 Backend

Das Backend ist dafür zuständig, einerseits das Token zu vergeben und andererseits Action-Methoden vor unberechtigtem Zugriff zu schützen, indem das generierte Token überprüft wird.

3.1 Package

Als Grundlage dient das Paket **`Microsoft.AspNetCore.Authentication.JwtBearer`**, das daher installiert werden muss (NuGet, Konsole oder Package Manager Konsole):



3.2 Startup.cs

In Startup.cs muss dann ein JwtBearer registriert werden.

3.2.1 ConfigureServices

Dieser braucht einen Key, der aus einem privaten Schlüssel erzeugt wird.

```
public void ConfigureServices(IServiceCollection services)
{
  string secret = "alkdsjlaksjdölkjlöaskelk";
  var key = Encoding.ASCII.GetBytes(secret);
  services.AddAuthentication(x =>
  {
    x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
  })
  .AddJwtBearer(x =>
  {
    x.RequireHttpsMetadata = false;
    x.SaveToken = true;
    x.TokenValidationParameters = new TokenValidationParameters
    {
      ValidateIssuerSigningKey = true,
      IssuerSigningKey = new SymmetricSecurityKey(key),
      ValidateIssuer = false,
      ValidateAudience = false
    };
  });

  services.AddCors(options =>...);
  services.AddMvc(options => options.EnableEndpointRouting = false);
}
```

3.2.2 Configure

In Configure muss die Middleware für Authentication in die Pipeline eingehängt werden:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
  if (env.IsDevelopment()) app.UseDeveloperExceptionPage();

  app.UseCors(myAllowSpecificOrigins);

  app.UseAuthentication();

  app.UseMvc();
}
```

3.3 IOptions

Das Secret ist oben noch in Startup.cs hart codiert. Vernünftiger wäre es, diese Information in **`appsettings.json`** zu speichern. Außerdem wird dieses Secret dann bei der Authentifizierung benötigt. Für diese Zwecke wird empfohlen, das über **`IOptions<>`** zu lösen.

3.3.1 appsettings.json

Zuerst also den Key hinterlegen:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Warning"
    }
  },
  "AllowedHosts": "*",
  "AppSettings": {
    "Secret": "MySpecialSecretKey"
  }
}
```

3.3.2 Klasse

Dann eine Klasse erzeugen, die genau jene Properties hat, die als Sub-Keys im JSON definiert sind – in unserem Fall also nur **`Secret`**:

```
public class AppSettings
{
  public string Secret { get; set; }
}
```

3.3.3 Startup.cs

Diese Klasse wird jetzt mit der entsprechenden Section von appsettings.json initialisiert und in der ServiceCollection mit **`Configure<>`** registriert:

```
public void ConfigureServices(IServiceCollection services)
{
  var appSettingsSection = Configuration.GetSection("AppSettings");
  services.Configure<AppSettings>(appSettingsSection);
  var appSettings = appSettingsSection.Get<AppSettings>();
  var key = Encoding.ASCII.GetBytes(appSettings.Secret);
  //string secret = "alkdsjlaksjdölkjlöaskelk";
  //var key = Encoding.ASCII.GetBytes(secret);
```

Dadurch kann jetzt eine Instanz von AppSettings als Singleton in allen Controller, Services,… über Dependency Injection angefordert werden.

3.4 AuthenticationController

Jetzt braucht man noch einen Controller, der über eine REST-Schnittstelle das Token an den Benutzer schickt.

3.4.1 UserService

Dazu soll zuerst ein Service **`UserService`** erstellt werden, der die Zugangsdaten eines Benutzers überprüft. Die User sind normalerweise natürlich in einer Datenbank hinterlegt, der Einfachheit halber sind sie hier direkt im Service hart codiert.

```
public class UserService
{
  private readonly List<User> users = new List<User>
  {
    new User{ Id=1, Username="hhuber", Role="Admin", Firstname="Hansi", Lastname="Huber", Password="quaxi"},
    new User{ Id=2, Username="sberger", Role="User", Firstname="Susi", Lastname="Berger", Password="12345"},
    new User{ Id=3, Username="pgruber", Role="Guest", Firstname="Pauli", Lastname="Gruber", Password="abcdef"},
  };

  public User Authenticate(string username, string password)
  {
    if (string.IsNullOrEmpty(username) || string.IsNullOrEmpty(password)) return null;

    var user = users.SingleOrDefault(x => x.Username == username);
    if (user == null) return null; //user does not exist

    return password == user.Password ? user : null;
  }
}
```

Nicht vergessen, das Service in Startup.cs zu registrieren!

3.4.2 Konstruktor

Im Konstruktor dann wie gewohnt alle Singeltons über Dependency Injection anfordern, die man braucht. In diesem Fall sind das das UserService sowie die AppSettings (weil man weiter unten dann das Secret braucht).

```
[Authorize]
[Route("[controller]")]
[ApiController]
public class AuthenticationController : ControllerBase
{
  private readonly UserService userService;
  private readonly AppSettings appSettings;

  public AuthenticationController(UserService userService, IOptions<AppSettings> appSettings)
  {
    this.userService = userService;
    this.appSettings = appSettings.Value;
  }
```

Hinweis: Im Beispiel wurde auch dieser Controller mit **`[Authorize]`** geschützt und nur die Authenticate-Methode mit **`[AllowAnonymous]`** für die Allgemeinheit freigegeben. Man könnte auch den gesamten Controller ungeschützt lassen.

3.4.3 Authenticate

In dieser Methode wird die eigentliche Authentifizierung durchgeführt. Zuerst werden die Zugangsdaten des Benutzers überprüft und daraus dann mittels des Secrets ein TokenString erzeugt.

```
[AllowAnonymous]
[HttpPost("authenticate")]
public ActionResult<AuthenticationDto> Authenticate([FromBody] UserDto userDto)
{
  var user = userService.Authenticate(userDto.Username, userDto.Password);

  if (user == null) return Unauthorized();

  string tokenString = CreateTokenString(user);

  return Ok(new AuthenticationDto
  {
    Id = user.Id,
    Username = user.Username,
    FirstName = user.Firstname,
    LastName = user.Lastname,
    Token = tokenString
  });
}
```

Dazu erzeugt man einen **`SecurityTokenDescriptor`** (https://docs.microsoft.com/en-us/dotnet/api/system.identitymodel.tokens.securitytokendescriptor?view=netframework-4.8&viewFallbackFrom=netcore-3.1):

```
private string CreateTokenString(User user)
{
  var tokenHandler = new JwtSecurityTokenHandler();
  var key = Encoding.ASCII.GetBytes(appSettings.Secret);
  var tokenDescriptor = new SecurityTokenDescriptor
  {
    Subject = new ClaimsIdentity(new[]
      {
        new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()),
        new Claim(ClaimTypes.Name, user.Username),
        new Claim(ClaimTypes.GivenName, user.Firstname),
        new Claim(ClaimTypes.Surname, user.Lastname),
        new Claim(ClaimTypes.Role, user.Role)
      }),
    Expires = DateTime.UtcNow.AddHours(4),
    SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key),
                                                SecurityAlgorithms.HmacSha256Signature)
  };
  var token = tokenHandler.CreateToken(tokenDescriptor);
  var tokenString = tokenHandler.WriteToken(token);
  return tokenString;
}
```

3.5 Testen

Das kann man jetzt mit RestClient oder Postman testen:

```
Send Request
POST http://localhost:5000/authentication/authenticate
Content-Type: application/json

{
  "username": "hhuber",
  "password": "quaxi"
}
```

Als Reply bekommt man ein eindeutiges Token:

```
HTTP/1.1 200 OK
Connection: close
Date: Mon, 02 Nov 2020 14:28:26 GMT
Content-Type: application/json; charset=utf-8
Server: Kestrel
Transfer-Encoding: chunked

{
  "id": 1,
  "username": "hhuber",
  "firstName": "Hansi",
  "lastName": "Huber",
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ
uYW1laWQiOiIxIiwidW5pcXVlX25hbWUiOiJoaHViZXIiLCJnaXZ
lbl9uYW1lIjoiSGFuc2kiLCJmYW1pbHlfbmFtZSI6Ikh1YmVyIiw
icm9sZSI6IkFkbWluIiwibmJmIjoxNjA0MzI3MzA2LCJleHAiOjE
2MDQzNDE3MDYsImlhdCI6MTYwNDMyNzMwNn0.BISJSNMR1gTSCLJ
BGaoPcNifvJbZO2OU7pbt5XAoUYk"
}
```

Bzw. bei Angabe eines falschen Accounts einen Reply „401 Unauthorized“:

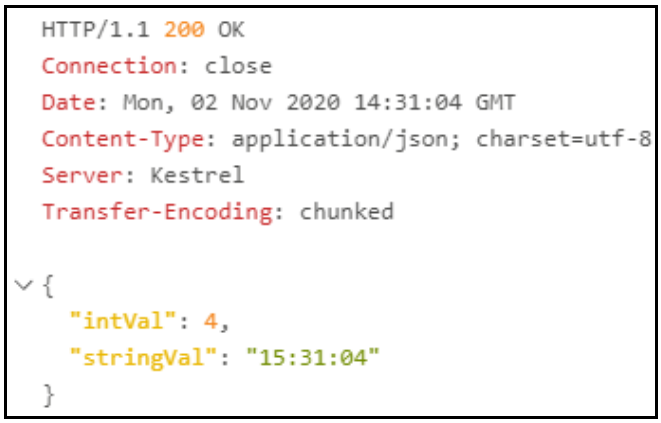
```
HTTP/1.1 401 Unauthorized
Connection: close
Date: Mon, 02 Nov 2020 14:30:02 GMT
Content-Type: application/problem+json; charset=utf-8
Server: Kestrel
Transfer-Encoding: chunked

{
  "type": "https://tools.ietf.org/html/rfc7235#section-3.1",
  "title": "Unauthorized",
  "status": 401,
  "traceId": "|20a8a388-4026e9315755a59b."
}
```

3.5.1 Mit Token

Fügt man dieses Token nun beim Request in den Header mit **`Authorization: Bearer myToken`** ein, sind die Aufrufe zulässig.

```
Send Request
GET http://localhost:5000/values/dummy
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6
```


HTTP/1.1 200 OK
Connection: close
Date: Mon, 02 Nov 2020 14:31:04 GMT
Content-Type: application/json; charset=utf-8
Server: Kestrel
Transfer-Encoding: chunked
{
"intVal": 4,
"stringVal": "15:31:04"
}

4 Frontend

Jetzt muss noch das Frontend so angepasst werden, dass Folgendes möglich ist

1. Authentication Service erzeugen
2. Authorisierung durchführen mit einer Login Page
3. Token speichern
4. Token bei allen Anfragen an das Backend mitsenden
5. Bestimmte Seiten von unbefugtem Zugriff schützen
6. Bei Zugriff auf geschützte Seiten automatisch auf eine Login-Seite umleiten

Der Einfachheit halber werden die verschiedenen Seiten nicht in separaten Module ausgelagert, sondern alles dem AppModule hinzugefügt.
Die Applikation selbst verwendet jedoch natürlich schon Routing.

4.1 Service

Die Authentifizierung sollte in ein eigenes Service programmiert werden (im Falle eines Moduls würde es anstelle des Core-Moduls ins Login-Modul notiert werden):

```
@Injectable({
  providedIn: 'root'
})
export class AuthenticationService {
  private urlBase = 'http://localhost:5000/authentication';

  constructor(private httpClient: HttpClient) { }

  public login(username: string, password: string): Observable<AuthenticationDto> {
    console.log(`AuthenticationService::login ${username}`);
    const url = `${this.urlBase}/authenticate`;
    return this.httpClient.post<AuthenticationDto>(url,
      { username: username, password: password });
  }
}
```

Hinweis: Das Passwort sollte nicht im Klartext übertragen werden, sondern verschlüsselt, z.B. mit einem MD5 Hash.

4.2 Login

Für das Login wird eine neue Komponente erzeugt, die über eine Route erreicht werden kann:

```
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'login', component: LoginComponent },
  { path: '**', redirectTo: '' }// otherwise redirect to home
];
```

Hier einfach Eingaben für Username und Passwort vorsehen (üblicherweise vernünftig gestylt…). Danach vorerst einmal einfach die Zugangsdaten oder eine Fehlermeldung anzeigen:

```
<h2>Login</h2>
<div>
  Benutzername: <input [(ngModel)]="username" />
</div>
<div>
  Passwort: <input [(ngModel)]="password" />
</div>
<div>
  <button (click)="login()">Login</button>
</div>
<div *ngIf="authenticationDto">{{authenticationDto|json}}</div>
<div *ngIf="errorMessage"
    class="error">Error: {{errorMessage}}</div>
```

Mit diesen Infos Authenticate im Backend aufrufen:

```
export class LoginComponent implements OnInit {
  username = 'hhuber';
  password = 'quaxi';
  authenticationDto: AuthenticationDto = null;
  errorMessage = '';

  constructor(private authenticationService: AuthenticationService) { }

  ngOnInit(): void {
    console.log(`LoginComponent::ngOnInit`);
  }

  public login(): void {
    console.log('LoginComponent::login');
    this.errorMessage = '';
    this.authenticationDto = null;
    this.authenticationService.login(this.username, this.password)
      .subscribe(
        x => this.authenticationDto = x,
        error => this.errorMessage = `${error.statusText}/${error.status} - User
      );
  }
}
```

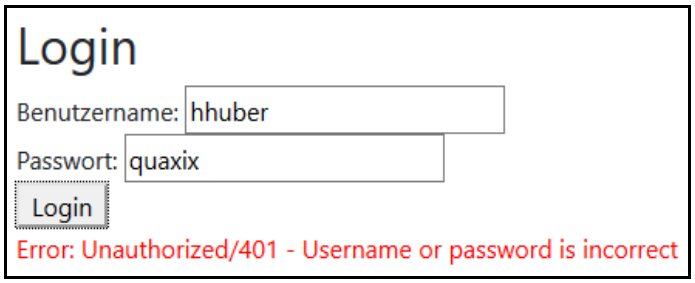
Login

Benutzername: hhuber

Passwort: quaxi

Login

{ "id": 1, "username": "hhuber", "firstName": "Hansi", "lastName": "Huber", "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1laWQiOiIxIiwidW5pcXVlX bXIETSD9E8izVcFARyRLyAy8HlHhJpEi8BkuaZDlak" }



4.3 Token speichern

Das Token muss für alle weiteren Zugriffe gespeichert werden. Das erfolgt üblicherweise im **`SessionStorage`**.
Beim Logout wird der Eintrag wieder gelöscht.
Am besten speichert man die gesamte User-Information:

```
export class AuthenticationService {
  private urlBase = 'http://localhost:5000/authentication';

  constructor(private httpClient: HttpClient) { }

  public login(username: string, password: string): Observable<AuthenticationDto> {
    console.log(`AuthenticationService::login ${username}`);
    const url = `${this.urlBase}/authenticate`;
    return this.httpClient.post<AuthenticationDto>(url, { username: username, passw
      .pipe(
        tap(user => {
          if (user && user.token) {
            sessionStorage.setItem('currentUser', JSON.stringify(user));
          }
        })
      );
  }

  public logout(): void {
    console.log('AuthenticationService::logout');
    sessionStorage.removeItem('currentUser');
  }
}
```

4.4 Token verwenden – Interceptor

Da das Token bei jedem Request mitgesendet werden muss, bietet sich an, das zu automatisieren. Genau dazu sind Interceptors da. Diese gehören ins Core-Modul.

```
PS C:\_PR\Angular\AngularRouting> ng g interceptor core\Jwt
CREATE src/app/core/jwt.interceptor.ts (408 bytes)
```

```
@Injectable()
export class JwtInterceptor implements HttpInterceptor {

  constructor() { }

  intercept(request: HttpRequest<unknown>, next: HttpHandler): Observable<HttpEvent<unknown>> {
    const currentUser = JSON.parse(sessionStorage.getItem('currentUser'));
    if (currentUser && currentUser.token) {
      request = request.clone({
        setHeaders: {
          Authorization: `Bearer ${currentUser.token}`
        }
      });
    }
    return next.handle(request);
  }
}
```

Nicht vergessen, diesen bei den Providers im Core-Modul zu registrieren:

```
@NgModule({
  declarations: [],
  imports: [
    CommonModule
  ],
  providers: [
    { provide: HTTP_INTERCEPTORS, useClass: JwtInterceptor, multi: true },
  ]
})
export class CoreModule { }
```

4.4.1 Homepage

Damit kann man jetzt in der Homepage auch die geschützten REST-Services benutzen:

```
<button routerLink="/home">Home</button>
<button routerLink="/login">Login</button>
<button (click)="logout()">Logout</button>
<hr />
<router-outlet></router-outlet>
```

```
export class AppComponent implements OnInit {

  constructor(private authenticationService: AuthenticationService) { }

  ngOnInit(): void {
    console.log('AppComponent::ngOnInit');
  }

  logout(): void {
    this.authenticationService.logout();
    location.reload();
  }
}
```

4.5 Seiten schützen – Guards

Man möchte ja nicht nur das Backend vor nicht authorisiertem Zugriff schützen, sondern auch am Frontend gewisse Seiten nicht von Unbefugten aufrufen lassen.
Das kann man in Angular durch **Guards** sicherstellen. Genaueres über Guards kommt in einem eigenen Dokument. Aber das Wesentliche sollte auch so verständlich sein.
Guards gehören in ein Core Modul. Für unsere Zwecke braucht man einen CanActivate-Guard:

```
PS C:\_PR\Angular\AngularRouting> ng g guard core\Auth
? Which interfaces would you like to implement? (Press <space> to select
>(*) CanActivate
 ( ) CanActivateChild
 ( ) CanDeactivate
 ( ) CanLoad
```

Die entstandene Klasse sieht so aus und implementiert das Interface CanActivate:

```
export class AuthGuard implements CanActivate {
  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean | U
    return true;
  }
}
```

Der Rückgabewert der einzigen Methode **canActivate** diese Interfaces gibt an, ob die Navigation auf die Seite, die mit diesem Guard geschützt wird (siehe weiter unten), erlaubt ist.

4.5.1 Login status prüfen

Die Methode canActivate jetzt also so umbauen, dass überprüft wird, ob im **SessionStorage** die User-Information gespeichert ist. Falls nicht, leitet man auf die Login-Seite um. Das geht wie gewohnt mit dem Router-Objekt. Idealerweise gibt man dabei als Parameter mit, auch welche Seite man ursprünglich navigieren wollte:

```
export class AuthGuard implements CanActivate {

  constructor(private router: Router) { }

  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | U
    if (sessionStorage.getItem('currentUser')) return true;
    this.router.navigate(['/login'], { queryParams: { returnUrl: state.url } });
    return false;
  }

}
```

4.5.2 Return url

In der Loginpage navigiert man anhand er returnUrl im Erfolgsfall auf die Ursprungsseite zurück:

```
ngOnInit(): void {
  console.log(`LoginComponent::ngOnInit`);
  this.authenticationService.logout(); // reset login status
  this.returnUrl = this.route.snapshot?.queryParams?.returnUrl || '/';
}
```

```
public login(): void {
  console.log('LoginComponent::login');
  this.errorMessage = '';
  this.authenticationDto = null;
  this.authenticationService.login(this.username, this.password)
    .subscribe(
      x => {
        this.authenticationDto = x;
        this.router.navigate([this.returnUrl]);
      },
      error => {
        console.dir(error);
        this.errorMessage = `${error.statusText}/${error.status} - Use
      });
}
```

4.5.3 Seite schützen

Um eine Seite vor unbefugtem Zugriff zu schützen, gibt man in app-routing.module.ts für die entsprechende Route dann den gewünschten Guard an.

```
const routes: Routes = [
  { path: '', component: HomeComponent, canActivate: [AuthGuard] },
  { path: 'login', component: LoginComponent },
  { path: '**', redirectTo: '' }// otherwise redirect to home
];
```

Auch hier gilt wieder das DRY-Prinzip: Man schreibt den Überprüfungscode in eine eigene Klasse und weist diese auf beliebige Seiten (bzw. eigentlich Routen) zu.

5 Rollen prüfen

Zum Abschluss soll noch gezeigt werden, wie man auf die Anmeldeinformationen am Backend zugreifen kann, um z.B. abzufragen, ob der Benutzer eine bestimmte Rolle einnimmt.

5.1 AdminController

Der Controller könnte dann so aussehen (er gibt aktuell nur Dummy-Daten zurück):

```
[Route("[controller]")]
[Authorize]
[ApiController]
public class AdminController : ControllerBase
{
  [HttpGet("MainData")]
  public AdminDummyDataDto GetMainData()
  {
    Console.WriteLine($"AdminController::{MethodBase.GetCurrentMethod().Name}");
    var userAuthInfo = this.GetUserAuthInfo();
    var reply = new AdminDummyDataDto();
    if (!userAuthInfo.IsAdmin) reply.ErrorMessage = "Only allowed for Admins!";
    return reply;
  }

}
```

Mit folgendem DTO:

```
public class AuthenticationReplyDto
{
  public bool IsOk { get; set; } = true;

  private string errorMessage;
  public string ErrorMessage
  {
    get => errorMessage;
    set
    {
      errorMessage = value;
      IsOk = false;
    }
  }

}
```

5.1.1 UserAuthInfo

Dazu braucht man die Authentifizierungsdaten des Benutzers. Das lagert man am besten in eine Extensionmethode aus:

```
public static UserAuthInfo GetUserAuthInfo(this ControllerBase controller)
{
  var claimsIdentity = controller.HttpContext.User.Identity as ClaimsIdentity;
  string role = claimsIdentity?.FindFirst(ClaimTypes.Role)?.Value ?? "Unknown";
  return new UserAuthInfo
  {
    Username = claimsIdentity?.FindFirst(ClaimTypes.Name)?.Value,
    Id = int.Parse(claimsIdentity?.FindFirst(ClaimTypes.NameIdentifier)?.Value),
    Role = role,
    Firstname = claimsIdentity?.FindFirst(ClaimTypes.GivenName)?.Value,
    Lastname = claimsIdentity?.FindFirst(ClaimTypes.Surname)?.Value
  };
}
```

5.2 Admin page

Die Verwendung könnte dann so aussehen:

```
export class AdminComponent implements OnInit {
  mainData: AdminDummyDataDto;

  constructor(private adminService: AdminService) { }

  ngOnInit(): void {
    console.log('AdminComponent::ngOnInit()');
    this.adminService.getMainData().subscribe(x => this.mainData = x);
  }
}
```

```
Message: {{mainData?.errorMessage}}
<h2 *ngIf="mainData?.isOk">Alles Ok</h2>
```

Wie gewohnt in app-routing.module.ts eintragen:

```
const routes: Routes = [
  { path: '', component: HomeComponent, canActivate: [AuthGuard] },
  { path: 'admin', component: AdminComponent, canActivate: [AuthGuard] },
  { path: 'login', component: LoginComponent },
  { path: '**', redirectTo: '' }// otherwise redirect to home
];
```

Ist jetzt kein Admin eingeloggt, wird das entsprechend angezeigt: