

Autentisering av användare

Hur skapas en säker inloggning i en fullstack applikation?

SUP-konferens 2022, Track A, fredag 4 mars, kl. 14:30

Plats: Örebro Universitet, T131

Grupp 17: Fabian Ahlqvist, Jonas Arvidson, Ludwig Ersson,

Maria Järventaus Johansson, Filip Koernig, Viggo Lagerstedt Ekholm

Innehållsförteckning

Inledning	2
JWT	3
Alternativ för att lagra tokens i webbläsaren	4
Http Cookies.....	5
localStorage och sessionStorage.....	5
Genomgång av ett login-anrop via en fullstack-app.....	8
1. Frontend – Login sidan.....	8
2. Frontend – Authentication Service	9
3. Backend – Authentication Controller.....	10
4. Backend – ApplicationUser Repository.....	11
5. Backend – Authentication Controller.....	11
6. Backend – Authentication Controller.....	12
7. Backend – AppSettings.json.....	13
8. Frontend – Login sidan.....	13
9. Frontend – AuthContextProvider.....	14
10. Frontend – Hook useAuth	15
11. Frontend – Hook useLocalStorage	16
12. Frontend – Hook useAuth	17

Inledning

Det kan ses som en självklarhet att det ska vara enkelt och säkert för en användare att logga in i en applikation. För en utvecklare som ska skapa ett sådant system är detta dock ett flertal moment som är långt ifrån enkla, speciellt om det gäller en webbapplikation som består både av backend och frontend. För att autentisera en användare krävs det att backend och frontend kan kommunicera med varandra på ett säkert sätt som förhindrar att andra kan få åtkomst till uppgifter som skulle tillåta dem att utge sig för att vara någon annan än de är.

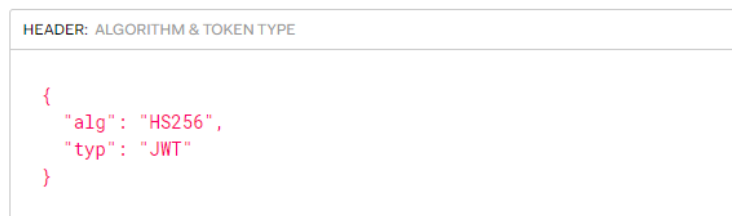
Det finns inte ett "rätt sätt" när det kommer till val av autentisering, utan utvecklare får välja det som passar deras applikation bäst. I detta material kommer olika alternativ för att autentisera en användare beskrivas samt redogöra olika nackdelar och fördelar gällande de olika alternativen.

JWT

Ett sätt att överföra information mellan olika parter är att använda sig av *JSON Web Token (JWT)*. I en JWT förs data över i form av ett JSON objekt. Den data som överförs i en JWT signeras digitalt vilket innebär att den kan verifieras och därför kan anses vara trovärdig. Signeringen av JWT sker genom krypteringsalgoritmer som, HMAC, RSA eller ECDSA (<https://jwt.io/introduction>).

JWT:s används vanligtvis för utbyte av information mellan olika parter eller för att autentisera en användare. Av dessa två scenarion är det autentisering som är det vanligaste användningsområdet. Med hjälp av JWT:s går det att behörighetsstyra användare genom att endast användare med viss behörighet har tillgång till olika *routes* och *services*. En JWT består av tre olika komponenter, en *header*, *payload* och *signature* (<https://jwt.io/introduction>).

Headern är oftast uppdelad i två delar, en del som beskriver vilken typ av token som används samt en del som beskriver vilken krypteringsalgoritm som används.



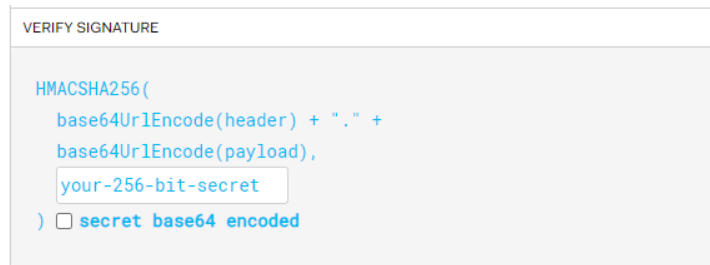
Figur 1. Visar en JWT Header med dess krypteringsalgoritm och typ av token (auth0, u.å)

Payloaden består av *claims*, vilket är information om t.ex. en viss användare. Claims kan i sin tur delas upp i tre olika kategorier, *registered*, *public* och *private*. Registered claims är sådana som är fördefinierade och som rekommenderas för alla som väljer att använda sig av JWT i sin applikation. Public claims är generella och kan användas fritt av den som valt att använda sig av JWT:s, exempel på detta är en användares mejl eller namn. Private claims är mer specifika och kan exempelvis vara en persons anställnings-id (<https://techdocs.akamai.com/api-gateway/docs/json-web-token-jwt-val#jwt-claims>).



Figur 2. Visar en JWT Payload innehållande subject, name och "issued at" (auth0, u.å)

En signature används för att verifiera en token och att den inte har ändrats någonstans. Om till exempel en token har signerats med en privat nyckel så går det också att verifiera vem det är som har skickat token. För att skapa en signature ska headern och payloaden **krypteras** med hjälp av en *secret key* som finns på serversidan (backend).



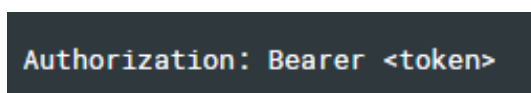
Figur 3. Visar signeringsdelen på jwt.io (auth0, u.å)

Resultatet av en JWT som har en kodad header och payload och är signerad secret key är en sträng som kan delas upp i tre olika delar. De tre olika komponenterna separeras med hjälp av en punkt. I bilden nedan är headern den översta delen (röd), payloaden mittersta delen (lila) och den understa delen är signaturen (blå). Med hjälp av signaturen går det att se om en JWT har blivit ändrad och på det sättet avgöra om den är giltig.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c

Figur 4. Visar en fullständig JWT, innehållande header, payload och signatur-information (auth0, u.å)

När en användare loggar in i ett system som använder sig av JWT så returneras en JWT tillbaka till den användaren. När en användare ska autentisera sig så skickas en JWT i "Authorization" headern för alla de anrop som sker mot WebApi servern. Authorization headern används för att bidra med referenser för att autentisera en användare, med användning av ett så kallat *Bearer schema*. Bearer scheme används för att meddela att en användare med en viss token ska få viss tillgång till ett system (<https://jwt.io/introduction>).



Figur 5. Visar ett Authorization Bearer-scheme (auth0, u.å)

Alternativ för att lagra tokens i webbläsaren

Den token som en användare får returnerad från backend servern vid inloggning kan lagras i användarens webbläsare via bland annat *Http Cookies*, *sessionStorage* eller *localStorage*. Detta gör det möjligt att skicka med en token för bland annat autentisering när användaren gör ett anrop på applikationen. De olika sätten att lagra en token på webbläsaren skiljer sig från varandra både i hur de

fungerar samt när det gäller säkerhetsaspekter. Nedan följer en redovisning av tre av de vanligaste sätten att spara tokens i webbläsaren.

Http Cookies

Http Cookies är en bit av information som sparas på en användares dator av en webbsida via en webbläsare. Anledningen till att man använder sig av *cookies* är för att kunna skapa en mer personanpassad upplevelse för en användare när den besöker en webbsida. Cookies kan bland annat innehålla en användares preferenser eller inputs (exempelvis användarnamn) som användaren skrivit in. Med hjälp av cookies skickas data fram och tillbaka i varje Http-anrop som görs i en applikation. Livslängden på en cookie beror på vilken typ av cookie det är. Session cookies har inget utgångsdatum då de utgår så fort som webbläsaren stängs ned. Dessa kan användas för att autentisera en användare när den navigerar runt på en webbsida så att användaren slipper autentisera sig vid varje anrop. Den andra typen är persistent cookies vars utgångsdatum ställs in manuellt. Dessa kan användas för att spara en användares preferenser så att de finns kvar nästa gång användaren surfar in på webbsidan.

<https://developer.mozilla.org/en-US/docs/Glossary/Cookie>.

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies>.

Om man använder sig av cookies kan man flagga dem som *HttpOnly*. Det innebär att data som lagras i cookien inte kan nås via JavaScript. Alltså kan inte en angripare komma åt data i en cookie via *Cross-Site Scripting* (XSS). XSS innebär bland annat att en part genom skadliga skript kan få tag i en JWT som finns sparad hos en annan användares Web Storage. Det går enbart att komma åt data i en HttpOnly cookie via backend-applikationen. Dock är det möjligt att via en *XSS + Cross-Site Request Forgery* (CSRF) komma åt cookien utan att kunna läsa dess data. Det innebär att med hjälp av cookien kan angriparen skicka ett anrop som kräver data som lagras i den aktuella cookien. Alltså kan en angripare inte komma åt data som lagras i cookien, men kan i stället använda den för att göra anrop mot backend.

<https://auth0.com/blog/secure-browser-storage-the-facts/>.

localStorage och sessionStorage

Till skillnad från cookies så behöver inte data skickas med varje Http-anrop när man använder sig av local eller sessionStorage. Både localStorage och sessionStorage är mekanismer i *Web Storage API* som låter webbläsaren spara key/value-par. Rent praktisk är det ett Storage objekt som sparas när man väljer att spara något i av tillvägagångssätten. I Storage objektet sparas värdet ihop med en nyckel och oavsett vilken datatyp som nyckel eller värdet är innan det sparas som ett Storage objekt konverteras de till strängar när det väl sparas ner.

När ett dokument laddas in i en webbläsare skapas en unik sessionStorage för den sidan i den aktuella fliken. Sessionen för sidan är sedan endast giltig i den specifika fliken. Öppnas samma sida i en ny flik skapas också en ny tom sessionStorage. Data som lagras i sessionStorage är även specifik för protokollet för den sidan. T.ex. sparas data som nås via HTTP inte i samma sessionStorage som data hämtad via HTTPS. sessionStorage töms på data när sidan eller fliken stängs ned men överlever om sidan bara uppdateras. <https://developer.mozilla.org/en-US/docs/Web/API/Window/sessionStorage>.

Tillskillnad från ovannämnda sessionStorage så finns det ingen utgångstid för hur länge information sparas i localStorage. Den informationen som sparas i localStorage finns kvar där tills dess att den tas bort manuellt eller programmatiskt. Detta ger fördelen att även om en användare stänger ner sin webbläsare så kan information, som till exempel en token sparas och nästa gång användaren går till samma webbplats så behöver denne inte logga in igen.

Nedan visas det hur en utvecklare kan gå till väga för att spara något i localStorage (samma sätt kan användas för att spara något i sessionStorage förutom att i stället för local så skrivs session).

```
localStorage.setItem('myCat', 'Tom');
```

Figur 6. För att spara något i ett Storage objekt används metoden setItem där nyckeln är det första argumentet följt av värdet som ska sparas.

```
const cat = localStorage.getItem('myCat');
```

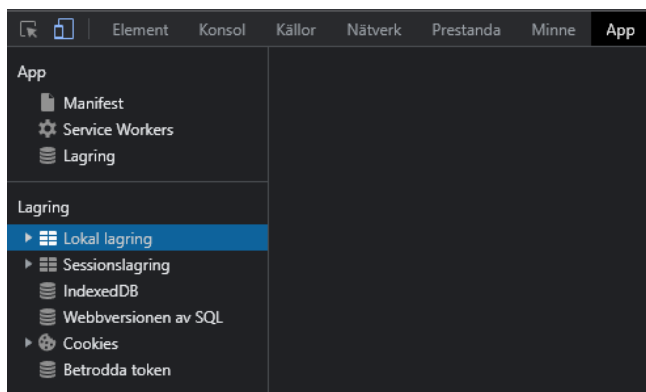
Figur 7. Genom att kalla på metoden getItem med nyckeln som argument går det att hämta det sparade värdet.

```
localStorage.removeItem('myCat');
```

Figur 8. Vill man ta bort ett specifikt sparad värde kan man ange nyckeln som den är sparad utefter.

```
localStorage.clear();
```

Figur 9. Vill man rensa hela minnet kan man tömma det med hjälp av clear.




Figur 10. Genom att gå in i DevTools och sedan navigera till App (Application) så går det sedan att se vad som finns sparad i lokal lagring (localStorage) eller sessionslagring (sessionStorage).

Både local eller sessionStorage är populära och enkla sätt att använda sig av för att lagra information i webbläsaren. Detta speglas också av att båda sätten är kompatibla med de mest använda webbläsarna.

	🖥️						📱						📋
	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	WebView Android	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet	Deno
localStorage	4	12	3.5	8	10.5	4	37	18	4	11	3.2	1.0	1.16 ★ ▼

Figur 11. Visar kompatibla webbläsar-versioner för localStorage.

	🖥️						📱						📋
	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	WebView Android	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet	Deno
sessionStorage	5	12	2	8	10.5	4	37	18	4	11	3.2	1.0	1.10

 Full support

Figur 12. Visar kompatibla webbläsar-versioner för sessionStorage.

Trots sin popularitet och användbarhet så finns det nackdelar med att använda sig av local eller sessionStorage. Både localStorage och sessionStorage erbjuder isolering av data genom "Same Origin Policy". Detta betyder att en sida inte kan nå local eller sessionStorage tillhörande en annan sida. Detta lämnar dock data sårbar för XSS-attacker eftersom den skadliga koden från dessa attacker uppfattas som legitim kod som kommer från den aktuella sidan. I och med att data i sessionStorage bara gäller för en specifik flik skyddas detta mot vissa typer av attacker. Till exempel om något sparats i en fliks sessionStorage och användaren sedan blir lurad att klicka på en extern länk innehållande skadlig XSS-kod. Den externa länken kommer då öppnas i en ny version av samma flik eller en helt ny flik vilket utelämnar möjligheten att komma åt data som sparats i den ursprungliga flikens sessionStorage. Detta minskar attackytan för kriminella men XSS-attacker kan dock genomföras utan externa länkar och för detta erbjuder sessionStorage inget skydd. <https://auth0.com/blog/secure-browser-storage-the-facts/>. När man använder sig av localStorage erbjuds det inte mycket skydd mot XSS-attacker. För att göra sin applikation säkrare mot XSS om man använder sig av localStorage kan man begränsa användningen av tredje-parts-bibliotek, ett annat sätt är att skriva säker kod, genom att alltid använda sig av validering för input som kan komma utifrån. Dessutom kan utvecklare välja att frånga localStorage och i stället kombinera sessionStorage med fingerprinting (https://cheatsheetseries.owasp.org/cheatsheets/JSON_Web_Token_for_Java_Cheat_Sheet.html).

Genomgång av ett login-anrop via en fullstack-app

1. Frontend – Login sidan

```
TS Login.tsx X
src > pages > authentication > TS Login.tsx > ...
1  import React from "react"
2  import { useAuthContext } from "../../State/Context/AuthContextProvider"
3  import { LoginRequest } from "../../service/Authentication"
4  import { AuthenticationCard } from "../../styles/Forms.styled"
5  import { Form } from "react-bootstrap"
6  import { Button } from "../../styles/Button.styled"
7  import { Center } from "../../styles/Align.styled"
8  import useValidate from "../../hooks/useValidate"
9  import Layout from "../../components/Layout"
10
11 const Login = () => {
12   const { login } = useAuthContext()
13
14   // #region FormValidation...
41
42   const Login = (event: any) => {
43     event.preventDefault()
44
45     const loginDto = {
46       username: usernameVal,
47       password: passwordVal,
48     }
49
50     LoginRequest(loginDto)
51       .then((jwtToken) => { login(jwtToken) })
52       .catch((error) => console.log(error))
53
54     // FormValidation on Username and Password
55     resetUsernameInput();
56     resetPasswordInput();
57   }
58
59   return (
60     <Layout>
```

1.1. Användaren har klickat på logga in via en React-frontend app, och kommer då till eventet på rad 42, där inloggningsfunktionen körs. En loginDto (Dto = Data Transfer Object) sätts upp med användarens inmatade användarnamn och lösenord. Därefter på rad 50 anropas LoginRequest, som är en importerad funktion från `../service/Authentication`, som tar emot loginDto-objektet som parameter.

2. Frontend – Authentication Service

```
TS Authentication.tsx X
src > service > TS Authentication.tsx > ...
1 import axios from "axios"
2 import { API } from "../components/Constants"
3 import { APIRequest } from "../Interceptor"
4 import { IUser } from "../User"
5
6 interface ILoginDto {
7   username: string
8   password: string
9 }
10
11 export async function LoginRequest(dto: ILoginDto) {
12   const request = axios.post(API + "/Authentication/login", dto)
13   return request
14     .then(response => response.data)
15     .catch(error => Promise.reject(error))
16 }
17
18 export async function GetLoggedInUser(): Promise<IUser | never> {
19   const request = APIRequest.get(API + "/Authentication/GetLoggedInUser")
20
21   return request
22     .then(response => response.data)
23     .catch(error => Promise.reject(error))
24 }
25
26 export async function Logout() {}
```

2.1. Väl inne i Servicen Authentication, så deklareras LoginRequest på rad 11. Den visar att den tar emot ett dto (Data Transfer Object), som har ILoginDto som interface (den ska ha samma schema som specificeras i interface-deklarationen som görs på rad 6).

2.2. På rad 12 deklareras en variabel som skickar en post-request med hjälp av Axios. Axios är en promise based¹ HTTP-klient för webbläsaren och node.js. Axios gör det enkelt att skicka asynkrona HTTP-förfrågningar till REST-slutpunkter och utföra CRUD-operationer. Axios genomför då en HTTP POST-förfrågan till API-url:en som hämtats ur konstanten på rad 2, samt går till Backend-API AuthenticationController samt metoden login och skickar då med datan från inloggnings formuläret, via dess specificerade DTO.

2.3. På rad 13 returneras sedan requesten. På rad 14 utförs dot notation(punktnotation) för att få fram svaret på den förfrågan som utfördes, genom att hämta datan som tillhör svaret. På rad 15, om det blir något fel, fångas det och ett *rejected promise* returneras.

¹ Promise based innebär att slutförandet (eller misslyckandet) av en asynkron operation alltid returnerar ett resulterande värde.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

3. Backend – Authentication Controller

```
AuthenticationController.cs
BackendAuthenticationDemo
BackendAuthenticationDemo.Controllers.AuthenticationController

1  using BackendAuthenticationDemo.Dtos;
2  using BackendAuthenticationDemo.Helpers;
3  using BackendAuthenticationDemo.Interfaces;
4  using BackendAuthenticationDemo.Models;
5  using Microsoft.AspNetCore.Authorization;
6  using Microsoft.AspNetCore.Mvc;
7  using Microsoft.IdentityModel.Tokens;
8  using System.IdentityModel.Tokens.Jwt;
9  using System.Security.Claims;
10 using System.Text;
11
12 namespace BackendAuthenticationDemo.Controllers;
13
14 [ApiController, Route(template: "api/[controller]")]
15 public class AuthenticationController : ControllerBase
16 {
17     private readonly IUnitOfWork _uow;
18     private readonly IConfiguration _config;
19
20     public AuthenticationController(IUnitOfWork uow, IConfiguration configuration)
21     {
22         _uow = uow;
23         _config = configuration;
24     }
25
26
27
28 [HttpPost, Route(template: "Login")]
29 public async Task<ActionResult<string>> LoginAsync(LoginDto dto)
30 {
31     if (ModelState.IsValid)
32     {
33         ApplicationUser user = await _uow.ApplicationUsers.Login(dto);
34         if (user.UserName == "User not found")
35             return NotFound(user.UserName);
36         if (user.UserName == "Wrong password")
37             return BadRequest(user.UserName);
38
39         var jwtToken = await CreateApiToken(user);
40         return Ok(jwtToken);
41     }
42
43     return BadRequest(dto);
44 }
45
46
47 GetLoggedInUser()
84
```

3.1. Eftersom vi nu fått en request till backend API:t, så ska vi också ta och förklara vad som händer här. På rad 14–15 och 28–29, så ser vi hur backends Route: "api/Authentication/Login" är den route/url som blivit anropad via ett HttpPost anrop med en modell av typen LoginDto som data. Därefter validerar vi datan på rad 31, genom att kontrollera att den är valid. Skulle den inte vara valid, så returneras ett HTTP 400 BadRequest, på rad 43 med dto:ns valideringsfel.

3.2. På rad 33, genomför vi själva anropet mot backend via vår Unit of Work (_uow), som anropar ApplivationUsersRepository och dess Login-metod, med den validerade dto:n. Om vi fått tillbaka "User not found" eller "Wrong password" så returneras HTTP 404 NotFound eller HTTP 400 BadRequest.

3.3. Här vi däremot fått tillbaka en korrekt user på rad 33, så körs metoden CreateApiToken (med den korrekta user), samt därefter returneras ett HTTP 200 OK, med den skapade jwtToken.

4. Backend – ApplicationUser Repository

```
ApplicationUserRepository.cs
BackendAuthenticationDemo
BackendAuthenticationDemo.Repositories.ApplicationUserRepository

1 using BackendAuthenticationDemo.Data;
2 using BackendAuthenticationDemo.Dtos;
3 using BackendAuthenticationDemo.Interfaces;
4 using BackendAuthenticationDemo.Models;
5 using Microsoft.AspNetCore.Identity;
6 using Microsoft.EntityFrameworkCore;
7
8 namespace BackendAuthenticationDemo.Repositories;
9
10 public class ApplicationUserRepository : RepositoryBase<ApplicationUser>, IApplicationUserRepository
11 {
12     protected new readonly ApplicationDbContext _dbContext;
13     public ApplicationUserRepository(ApplicationDbContext dbContext) : base(dbContext)
14     {
15         _dbContext = dbContext;
16     }
17
18     public async Task<ApplicationUser> Login(LoginDto dto)
19     {
20         var user = await _dbContext.AspNetUsers.FirstOrDefaultAsync(u => u.UserName.Equals(dto.UserName));
21         if (user == null)
22             return new ApplicationUser() { UserName = "User not found" };
23
24         var hasher = new PasswordHasher<ApplicationUser>();
25         var passwordHash = hasher.HashPassword(user, dto.Password);
26         PasswordVerificationResult passwordCheck = hasher.VerifyHashedPassword(user, user.PasswordHash, dto.Password);
27
28         if (passwordCheck.HasFlag(PasswordVerificationResult.Success))
29             return user;
30
31         return new ApplicationUser() { UserName = "Wrong password" };
32     }
33
34     GetAllUserInfoById(Guid userId)
35 }
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
```

4.1. Ifrån steg 3, rad 33, så hade Login-metoden i vårt ApplicationUserRepository anropats som då tar emot den validerade LoginDto:n. Där hämtas sedan användaren ifrån databas-kontextet på rad 20. Hittas inte användaren, så skickas en tom användare tillbaka med "User not found".

4.2. Rad 24–26, genomförs en hashning av det inmatade lösenordet, som sedan kontrolleras mot databasens sparade lösenords hash. Är PasswordVerificationResult, satt som Success, så returneras användaren i sin helhet. Annars returneras en tom användare med Wrong password.

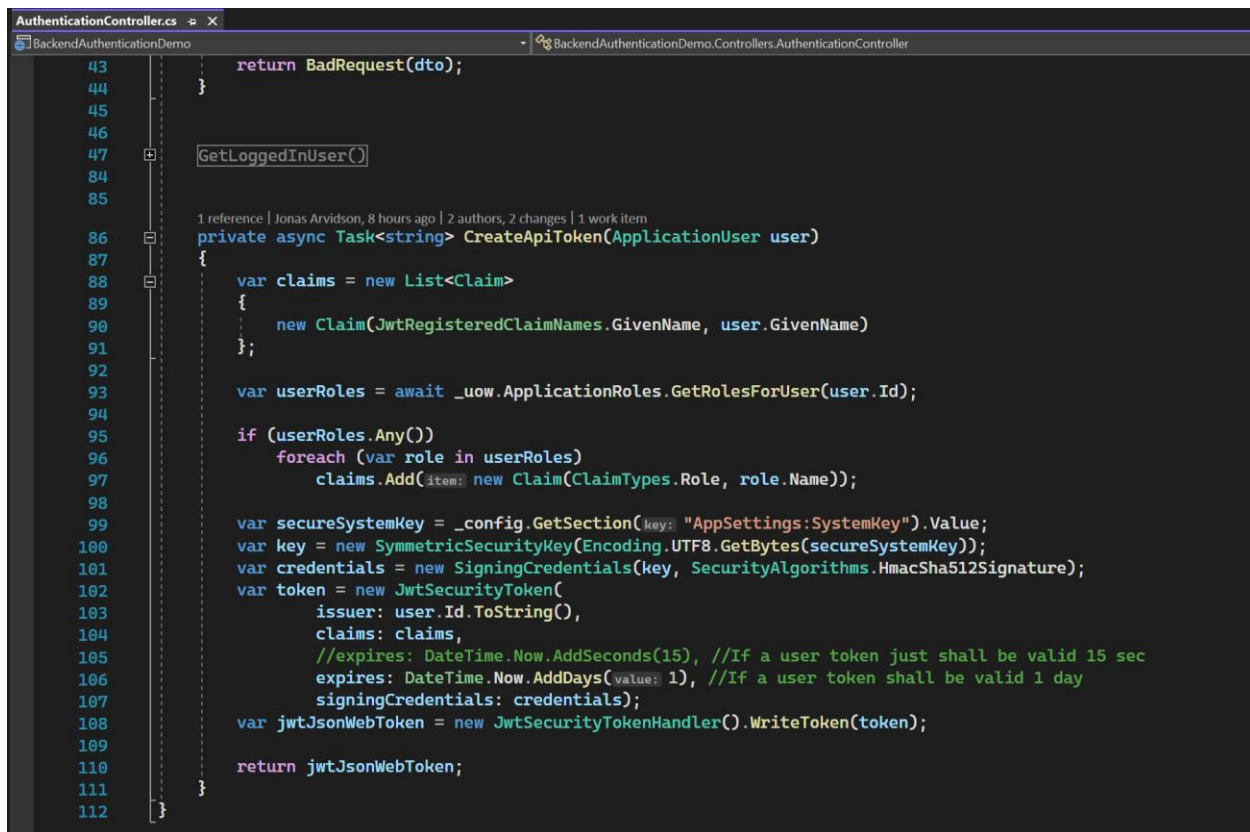
5. Backend – Authentication Controller

```
AuthenticationController.cs
BackendAuthenticationDemo
BackendAuthenticationDemo.Controllers.AuthenticationController

28 [HttpPost, Route("template: Login")]
29 public async Task<ActionResult<string>> LoginAsync(LoginDto dto)
30 {
31     if (ModelState.IsValid)
32     {
33         ApplicationUser user = await _uow.ApplicationUsers.Login(dto);
34         if (user.UserName == "User not found")
35             return NotFound(user.UserName);
36         if (user.UserName == "Wrong password")
37             return BadRequest(user.UserName);
38
39         var jwtToken = await CreateApiToken(user);
40         return Ok(jwtToken);
41     }
42
43     return BadRequest(dto);
44 }
```

5.1. Om vi sedan fått tillbaka en korrekt användare från rad 33 i vår controller, så körs rad 39, som skapar upp en JWT via CreateApiToken-metoden inne i samma controller.

6. Backend – Authentication Controller



```
43         return BadRequest(dto);
44     }
45
46     [HttpGet]
47     public async Task<string> GetLoggedInUser()
48     {
49         // ...
50     }
51
52     1 reference | Jonas Arvidson, 8 hours ago | 2 authors, 2 changes | 1 work item
53     private async Task<string> CreateApiToken(ApplicationUser user)
54     {
55         var claims = new List<Claim>
56         {
57             new Claim(JwtRegisteredClaimNames.GivenName, user.GivenName)
58         };
59
60         var userRoles = await _uow.ApplicationRoles.GetRolesForUser(user.Id);
61
62         if (userRoles.Any())
63             foreach (var role in userRoles)
64                 claims.Add(new Claim(ClaimTypes.Role, role.Name));
65
66         var secureSystemKey = _config.GetSection(key: "AppSettings:SystemKey").Value;
67         var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(secureSystemKey));
68         var credentials = new SigningCredentials(key, SecurityAlgorithms.HmacSha512Signature);
69         var token = new JwtSecurityToken(
70             issuer: user.Id.ToString(),
71             claims: claims,
72             //expires: DateTime.Now.AddSeconds(15), //If a user token just shall be valid 15 sec
73             expires: DateTime.Now.AddDays(value: 1), //If a user token shall be valid 1 day
74             signingCredentials: credentials);
75         var jwtJsonWebToken = new JwtSecurityTokenHandler().WriteToken(token);
76
77         return jwtJsonWebToken;
78     }
79 }
```

6.1. Här ser vi den privata CreateApiToken på rad 86 i vår AuthenticationController, med vår hämtade ApplicationUser från databasen via vårt Repository. Nu börjar skapandet av den jwtToken som sätts upp utifrån de standarder som finns för hur en jwt ska skapas.

6.2. Vi börjar med att skapa en ny lista av Claims på rad 88–91, som innehåller en ny Claim med användarens förnamn. Därefter hämtar vi alla användarens roller på rad 93. Om användaren har några roller (.Any()), så ska varje roll adderas till användarens ClaimType.Role med rollens namn.

6.3. När vi sedan har användarens namn och roller i vår token, så hämtar vi ut systemets säkerhetsnyckel från AppSettings:SystemKey, i detta fallet (se rad 13 i steg 16), därefter tar vi denna säkerhetsnyckeln och på rad 100, skapar vi en SymmetricSecurityKey av denna säkerhetsnyckel strängen.

Rad 101, skapar vi upp användarens signerade autentiseringsuppgifter (SigningCredentials), med hjälp av vår krypteringsnyckel och krypteringsmetoden Hmac Sha512.

6.4. Rad 102–107, skapas sedan den token som innehåller utfärdare (issuer), användarens anspråk (claims) i form av förnamn och roller, även när denna utfärdade token ska upphöra (expires), samt dess signerade autentiseringsuppgifter, så att token:en vid anrop till servern också kan valideras att dessa är korrekt utfärdade.

6.5. Rad 108, skrivs sedan vår token om till standard JWT-formatet och returneras på rad 110.

7. Backend – AppSettings.json

```
appsettings.json
Schema: https://json.schemastore.org/appsettings.json
1 {
2   "Logging": {
3     "LogLevel": {
4       "Default": "Information",
5       "Microsoft.AspNetCore": "Warning"
6     }
7   },
8   "AllowedHosts": "*",
9   "ConnectionStrings": {
10    "AppDbContext": "Server=. ;Database=BackendAuthDemoDb;Trusted_Connection=True;MultipleActiveResultSets=True"
11  },
12  "AppSettings": {
13    "SystemKey": "314ca2e724f74f68bfcae01ecc070153--aTopSecretSystemApiKey--"
14  }
15 }
16 }
```

7.1. Här ser vi i vår appsettings.json-fil, den databas-anslutningen som anropas på rad 10, samt vår AppSettings:SystemKey på rad 13 som används på rad 99 i steg 10, samt vid validering av token inne i Program.cs på rad 52.

7.2. Nu har vi nu genomgått hela backend anropet för inloggningen och skickat tillbaka en jwtToken från steg 10 rad 112 och steg 6 rad 40, så ska vi förklara vidare vad som sedan sker i frontend efter det att den strängen med jwtToken:en är skickad från backend till frontend.

8. Frontend – Login sidan

```
18 Login.tsx
src > pages > authentication > 18 Login.tsx > ...
1 import React from "react"
2 import { useAuthContext } from "../../State/Context/AuthContextProvider"
3 import { LoginRequest } from "../../service/Authentication"
4 import { AuthenticationCard } from "../../styles/Forms.styled"
5 import { Form } from "react-bootstrap"
6 import { Button } from "../../styles/Button.styled"
7 import { Center } from "../../styles/Align.styled"
8 import useValidate from "../../hooks/useValidate"
9 import Layout from "../../components/Layout"
10
11 const Login = () => {
12   const { login } = useAuthContext()
13
14   //region FormValidation...
15
16   const Login = (event: any) => {
17     event.preventDefault()
18
19     const loginDto = {
20       username: usernameVal,
21       password: passwordVal,
22     }
23
24     LoginRequest(loginDto)
25       .then(jwtToken => { login(jwtToken) })
26       .catch(error => console.log(error))
27
28     //FormValidation on Username and Password
29     resetUsernameInput();
30     resetPasswordInput();
31   }
32
33   return (
34     <Layout>
35
```

8.1. På rad 50 har vi nu fått en token eller ett fel returnerat från Backend-API:t. På den tokenen utförs .then via Axios på rad 51, som anger att jwtToken (den data som returnerats från rad 50) ska skickas med som variabel i funktionen login(). Denna funktion är instansierad som ett objekt av useAuthContext som login på rad 12. useAuthContext är importerad från "../../State/Context/AuthContextProvider".

9. Frontend – AuthContextProvider

```
TS AuthContextProvider.tsx X
src > state > context > TS AuthContextProvider.tsx > ...
1  import React, { createContext, useContext } from "react"
2  import useAuth from "../../hooks/useAuth"
3
4  interface Props {
5    children: JSX.Element
6  }
7
8  export const AuthContext =
9    createContext<ReturnType<typeof useAuth | null>>(null)
10
11 export const useAuthContext = () => useContext(AuthContext)!
12
13 function AuthContextProvider({ children }: Props) {
14   const auth = useAuth()
15   return <AuthContext.Provider value={auth}>{children}</AuthContext.Provider>
16 }
17
18 export default AuthContextProvider
```

9.1. På rad 11 ser vi useAuthContext deklareras. Detta görs via React Hooks² (som är ett sätt att spara property-värden utan att behöva skapa klasser). useContext tar emot AuthContext som parameter, vilket anger att AuthContext ska sparas som property-värde.

9.2. På rad 8 är AuthContext deklarerad, och React-funktionen createContext anropas och skapar ett kontext-objekt, av typen useAuth eller null.

² Information om React Hooks finner man på sidan: <https://reactjs.org/docs/hooks-reference.html#usecontext>

10. Frontend – Hook useAuth

```
TS useAuth.tsx X
src > hooks > TS useAuth.tsx > [🔍] default
1  import { useReducer, useEffect } from "react";
2  import useLocalStorage, { STORED_VALUES } from "../useLocalStorage";
3  import jwt from 'jwt-decode';
4  import { navigate } from "gatsby";
5  import { AuthReducer, initialState, State } from "../State/reducers/AuthReducer";
6
7  export interface JWT {
8      name: string;
9      given_name: string;
10     roles: string;
11     exp: number;
12     iss: string;
13 }
14
15 function useAuth() {
16     const [authState, authDispatch] = useReducer(AuthReducer, initialState);
17     const { value, setValue, remove } = useLocalStorage(STORED_VALUES.TOKEN, "");
18
19     useEffect(() => {
20         if (value) { login(value); } else { logout(); }
21     }, [])
22
23     const login = (responseToken: string) => {
24         const user = jwt<JWT>(responseToken);
25         authDispatch({
26             type: 'LOGIN_ACTION',
27             username: user.given_name,
28             id: user.iss
29         });
30         setValue(responseToken);
31         navigate("/admin/Start");
32     }
33
34     const logout = () => {
35         remove(STORED_VALUES.TOKEN);
36         authDispatch({ type: 'LOGOUT_ACTION' });
37         navigate("/");
38     }
39
40     const isLoggedIn = (): boolean => { return authState.isAuth; }
41
42     const getLoggedInUser = (): State => { return authState; }
43
44     return { login, logout, isLoggedIn, getLoggedInUser }
45 }
46
47 export default useAuth;
```

10.1. Funktionen useAuth deklarerar först två variabler. AuthState på rad 16, som används i de returnerade värdena isLoggedIn och getLoggedInUser. Detta görs genom useReducer som låter oss specificera hur vi vill uppdatera ett objekts property. Detta specificeras i AuthReducer.

10.2. På rad 17 deklareras en variabel för funktionen useLocalStorage, som importeras från "../useLocalStorage".

11. Frontend – Hook useLocalStorage

```
TS useLocalStorage.tsx X
src > hooks > TS useLocalStorage.tsx > ...
1  import { useState, useEffect } from "react"
2
3  export enum STORED_VALUES {
4    | TOKEN = "STORED_VALUES.TOKEN",
5  }
6
7  function getSavedValue(key: STORED_VALUES | string, initialValue: any) {
8    const savedValue = JSON.parse(localStorage.getItem(key) as string)
9    if (savedValue) return savedValue
10
11    if (initialValue instanceof Function) return initialValue()
12    return initialValue
13  }
14
15  function useLocalStorage(key: STORED_VALUES | string, initialValue: any) {
16    const [value, setValue] = useState(() => {
17      | return getSavedValue(key, initialValue)
18    })
19
20    useEffect(() => {
21      | localStorage.setItem(key, JSON.stringify(value))
22    }, [value])
23
24    const remove = (key: string) => {
25      | localStorage.removeItem(key)
26    }
27
28    return { value, setValue, remove }
29  }
30
31  export default useLocalStorage
32
```

11.1. useLocalStorage innehåller ett enum, "STORED_VALUES" och funktionerna getSavedValue och useLocalStorage. UseLocalStorage tar värdet vi har sparad som token på rad 4 genom "key" på rad 17, och sparar det som ett item i local storage på rad 21, tillsammans med value som hämtas från getSavedValue på rad 17, initialValue.

12. Frontend – Hook useAuth

```
TS useAuth.tsx X
src > hooks > TS useAuth.tsx > [0] default
1 import { useReducer, useEffect } from "react";
2 import useLocalStorage, { STORED_VALUES } from "../useLocalStorage";
3 import jwt from 'jwt-decode';
4 import { navigate } from "gatsby";
5 import { AuthReducer, initialState, State } from "../State/reducers/AuthReducer";
6
7 export interface JWT {
8   name: string;
9   given_name: string;
10  roles: string;
11  exp: number;
12  iss: string;
13 }
14
15 function useAuth() {
16   const [authState, authDispatch] = useReducer(AuthReducer, initialState);
17   const { value, setValue, remove } = useLocalStorage(STORED_VALUES.TOKEN, "");
18
19   useEffect(() => {
20     if (value) { login(value); } else { logout(); }
21   }, [])
22
23   const login = (responseToken: string) => {
24     const user = jwt<JWT>(responseToken);
25     authDispatch({
26       type: 'LOGIN_ACTION',
27       username: user.given_name,
28       id: user.iss
29     });
30     setValue(responseToken);
31     navigate("/admin/Start");
32   }
33
34   const logout = () => {
35     remove(STORED_VALUES.TOKEN);
36     authDispatch({ type: 'LOGOUT_ACTION' });
37     navigate("/");
38   }
39
40   const isLoggedIn = (): boolean => { return authState.isAuth; }
41
42   const getLoggedInUser = (): State => { return authState; }
43
44   return { login, logout, isLoggedIn, getLoggedInUser }
45 }
46
47 export default useAuth;
```

12.1. Efter att de två variablerna på rad 16 och 17 deklarerats i steg 10, så används `useEffect` på rad 19 för att avgöra om det finns ett värde för `value`, då funktionen `login` körs genom att skicka med `value`, eller om `logout` ska köras.

12.2. `login` definieras på rad 23. Konstanten `user`, på rad 24, anges vara en JSON Web Token (se tidigare förklaring på sidan 2–3). `authDispatch` uppdaterar state (värdena på properties),

12.3. På rad 30 sätts värdet att spara i local storage, och `setValue` anropas på rad 17 genom `useLocalStorage`.

12.4. På rad 31 används en Gatsby-funktion som navigerar användaren till rätt sida då den loggat in.

12.5. Om i stället en utloggning ska göras, anropas metoden på rad 34, och det värde som sparats i local storage raderas, och användaren navigeras till en utloggad sida.