Produktrapport

Agora

Titelblad

Deltagere:

Stefan Lynge Hvilsom

Projektnavn:

Agora

Dato:

13/11 2025

Skole:

Techcollege

Struervej 70

9220 Aalborg Ø

Vejledere:

Frank Rosbak

Lars Thise Pedersen

Underskrifter:

Indholdsfortegnelse

Indledning

Denne produktrapport beskriver udviklingen af et social medie løsning, bestående af et .NET baseret API og en web frontend i Svelte. Formålet er at give brugere et enkelt sted hvor de kan lave korte opslag, kommentere, like indhold og følge andre brugere. Rapporten gennemgår produktets overordnede mål, arkitektur og datamodel, sikkerhed (autentificering, /autorisation), centrale funktioner, samt driftsmæssige forhold (deployment /konfiguration). Fokus er på at skabe en skalérbar løsning, som samtidig er sikker. Løsningen er implementeret med ASP.NET Core(.NET9), EF Core (Entity Framework), SQL Server, JWT baseret login med httpOnly refresh-cookies, og CORS-opsætning til at sikre kommunikationen mellem API og frontend. Der benyttes også SignalR til at sikre ”realtime” dataoverførsel.

Læsevejledning

Formålet med denne produktrapport er at den dokumenterer produktet, dets funktionalitet, arkitektur, data og drift. Procesmæssige valg, ændringer i overvejelser og ikke-implementerede dele behandles i en separat procesrapport.

Målgruppe: Tekniske læsere (Vejleder/censor).

Sådan læses rapporten:

Overblik:

Start med indledning og beskrivelse af produkt, for at få overordnet ide om hvad produktet indeholder.

Teknik:

Arkitekturen gennemgåes hvor vi ser på lagdelingen (API, data, interfaces, services og EF) og hvordan disse integreres. Sikkerheden i projekter indeholder autorisation autentikation.

Vi gennemgår også databasen og relationerne imellem modellerne. Rapporten viser hvordan programmet er blevet konfigureret, og hvordan versionsstyring bliver brugt. Hertil vil der også blive vist hvordan de forskellige test er blevet lavet.

Kravspecifikation

- Id: indikerer rækkefølgen af hvornår specifikationen skal laves
- Kategori: hentyder til hvilken del af produktet denne specifikation er en del af
  - API + specifikation
  - Front
  - DB
- Test: Dette er et felt hvor der beskrives hvilken slags test der er udført.
- Beskrivelse: Kort beskrivelse af hvad specifikationen indeholder.

Id	Kategori	Test	Beskrivelse
1	API Models	**Unit test:** Opret en bruger, slette brugeren igen, oprette en chat og slette igen	Models indeholder: • Users • Comment • Posts • Likes • Followers • Roles • UserRoles • RefreshTokens
2	API DatabaseContext		Databasecontext gør brug af Entity Framework til at opsætte modellerne til databasen
3	API Data Transfer Object (DTO)		Opret DTO'er til modellerne
4	API Interfaces		Oprette interfaces til modellerne så API er mere modulær og det vil være nemmere at udvide den i fremtiden

5	API Utilities		Lav en passwordhasher til at hashe passwords.
6	API JWTtokens		Gør så API kan modtage og oprette JWT tokens
7	DB		Setup databasen med din EF så databasen er konfigureret
8	API Services		Opret services til dine interfaces
9	API Controllers	**Kontrakt test:** Oprette en bruger og login og logud i terminalen	Oprette controllers til alle services
10	Front LoginPage	**Integration test:** login med en bruger og tjek at den for JWT token	Oprette en loginside hvor man kan indsætte email og password, samt knap til login og link til registrer bruger
11	Front RegisterPage	**Integration test:** Registrer en bruger og tjek at den bliver oprettet i databasen	Oprette input til fornavn efternavn email og password
12	Front Dashboard	**Integration test:** Efter login, opret en chat og en post og se om de bliver oprettet i databasen	Oprette input til at oprette en ny chat, samt designe andre chats fra andre brugere hvor der skal være et input til at skrive en comment og en knap til at post den comment
13	Front Data download		Oprette en knap hvor brugeren kan downloade sin data
14	API Security	**GDPR complience test:** Test automatisk sletning af data efter 6 måneder	Implementer en autmatisk sletning af data, for inaktive brugere
15	Front Cookie		Lave en cookie banner så brugerne selv kan vælge at

			acceptere cookies som man skal ifølge GDPR
16	DB Sikkerhed		Sikre at alt data befinder sig i EU
17	API Right to be forgotten		Lav et API kald der kan slette al bruger data efter ønske fra bruger
18	API Sikkerhed		Vær sikker på at der kun er den minimale mængde data der bliver sendt med hvert API kald
19	Front		Lav en side hvor brugeren kan ændre sine cookie samtykkevalg
20	API SignalR		Opsætte API 'et til et køre med SignalR så der kan vises opslag og kommentarer med det samme de bliver postet

Produktdokumentation

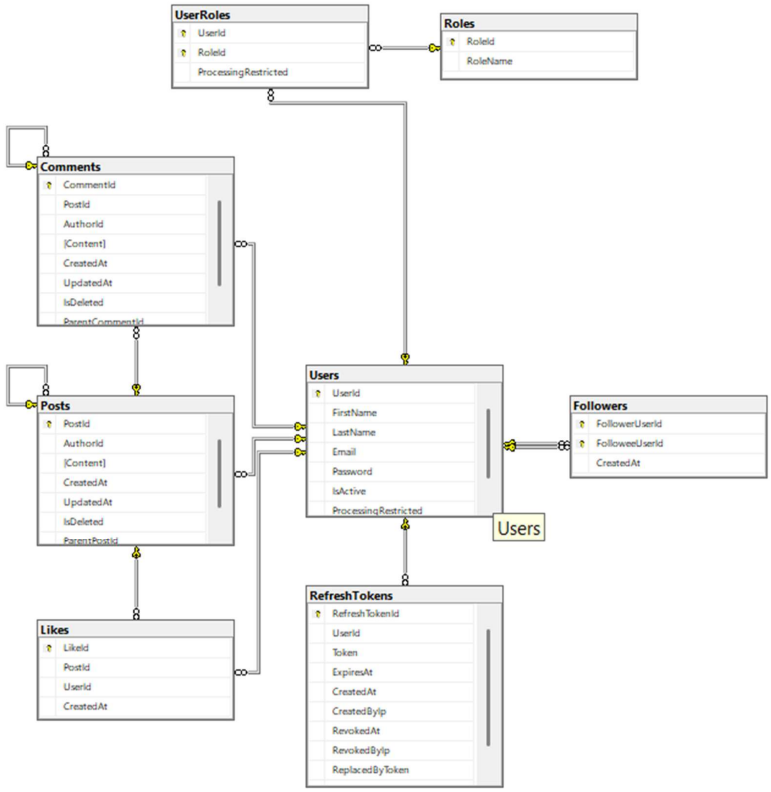
Versionsstyring

I projektet er git blevet brugt i samarbejde med GitHub, til at styre ændringer løbende. Der blev lavet et issue til alle de specifikationer der blev lavet i API'en, disse issues har fået adskillige subissues under sig, hvor der er blevet lavet en gren til hvert subissue. Dette har gjort at der kun er blevet lavet mindre ændringer i systemet, som gør at nemmere kan lave et "rollback" hvis der er fejl. Hver af disse "rollback" point er også kaldet for commits Hver af disse commit har også fået en tydelig beskrivelse, så man kan se hvad det indeholder. Når der bliver pushet afsted, er der blevet lavet en pull forespørgsel til main branch, som der skal merge selvstændigt på github. På denne måde kan det sikre sig at alle læser alle commits igennem inden de bliver merged en branch med main. Når projektet skal "deployes" sker dette kun via main branchen, så det altid er en gennemtestet branch som i dette tilfælde er blevet pushed til Azure Deployment Center via GitHub Actions.

Konfigurationen af hemmeligheder er sat op igennem Azure App Service, hvor forbindelsesstrenge til Databasen og JWT-hemmeligheder er opbevaret, så de ikke ligger til skue for alle og en hver. Den appsetting.json som findes på GitHub indeholder kun test data, da det blev kørt lokalt.

Database

ER-Diagram



Databasediagrammet, viser alle relationerne mellem modellerne.

Særligt bemærkes de selv-refererende relationer i Posts og Comments. Disse loops er lavet med vilje, fordi både en post og en kommentar kan have en ”parent”.

Dette betyder at en PostModel kan pege på en anden PostModel via et ParentPostId f.eks. en repost eller et svar på en post. Og det samme gør sig gældende for en CommentModel.

Fordelen er at der kan bygges hierarkier uden ekstra tabeller. EF-core konfigurationen gør relationerne valgfrie. Så en post/kommentar kan stå alene eller indgå som et ”barn”.

Der er også andre centrale relationer bla. User -> Role via UserRole som er en many-to-many relation. Der er også Follower som også er en selv-refererende model med many-to-many relation mellem brugere (følgere og følges af). Like forbinder en bruger og en post som er unik pr. bruger/post.

Program

Arkitektur

Systemarkitektur og Designprincipper

Agora systemet følger en lagdelt arkitektur baseret på SOLID principperne, som sikrer en vedligeholdelsesvenlig, testbar og skalérbar kodestruktur. Arkitekturen er opdelt i tydeligt adskilte lag med tydeligt definerede ansvarsområder.

SOLID principper

Single Responsibility Principle (SRP)

Hver klasse i systemet har et enkelt og veldefineret ansvarsområde. Controllere håndterer udelukkende http logik, services indeholder forretningslogik, og data access laget varetager databaseoperationer gennem AppDbContext. Modellerne repræsenterer kun datastrukturen uden at indeholde forretninglogik.

Open/Closed principle (OCP)

Systemet er designet til at være åbent for udvidelse, men lukket for modifikation. Gennem brug af interfaces kan ny funktionalitet tilføjes uden at ændre eksisterende kode. Der kan f.eks. tilføje nye autentifikations metoder uden at man påvirker den nuværende loginmetode.

Liskov Substitution Principle (LSP)

Systemets interface hierarki er designet således, at implementerende klasser kan erstatte hinanden uden at bryde systemets funktionalitet. Dette sikrer at vi har en konsistent adfærd på tværs af forskellige implementeringer.

Interface Segregation Principle (ISP)

Interfaces er opdelt i små, specifikke kontraker i steder for store, generelle interfaces. Dette gør at der kan reduceres afhængigheder og gør systemet mere fleksibelt. For eksempel er der separate interfaces for brugerhåndtering, autentifikation og Post håndtering.

Dependency Inversion Principle (DIP)

Højniveaumoduler afhænger ikke af lavniveaumoduler, men begge afhænger af abstraktioner. Dette opnås gennem depency injection, hvor konkrete implementeringer injiceres i constructors baseret på interface abstraktioner.

Lagdelt arkitektur

Systemet er organiseret i fire primære lag:

- **Presentation Layer:** Controllere der håndterer http forespørgsler og svar
- **Business Layer:** Services der indeholder forretningslogik og domæneregler
- **Data Accass Layer:** AppDbContext og Entity Fromwaor der håndterer databaseoperationer, samt modeller der repræsenterer datastrukturen
- **Domain Layer:** Modeller og DTO'er der definerer systemets domænemodel

.NET

I .NET laget eksponeres der en tynd controller, som udelukkende varetager systemets REST-endpoint. REST står for Representational State Transfer, der er en stateless arkitektur som betyder at serveren ikke gemmer klientsessionen, og hver forespørgsel indeholder al nødvendig information. REST indeholder også den kendte CRUD, som står for CREATE, READ, UPDATE, DELETE, som er de kendte HTTP (hypertransfer text protocol). Vores API gør så brug af services der varetager al API'ens forretningslogik, som gør brug af veldefinerede interfaces. Data sendes aldrig direkte fra API'ens modeller, hvor alle modellernes properties er, men der laves derimod noget der hedder DTO(Data Transfer Object). Systemet bruger herefter Automapper som mapper DTO'erne til modellerne, for at bevare dataminering så det kun er den nødvendige data der bliver send afsted.

Entity Framework

Entity Framework Core, fungere som systemets ORM(Object Relationel Mapper), der oversætter mellem Objektorienteres kode i .NET og den relationelle database. I stedet for at

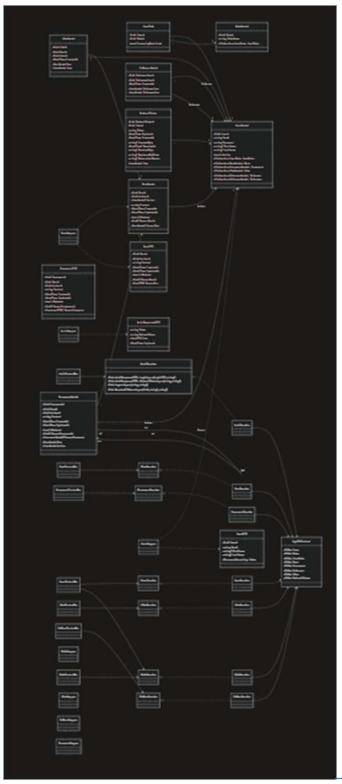
skrive SQL-kode manuelt, lader EF Core systemet arbejde med databaseobjekter som om de var almindelige C# klasser.

Hvad er en ORM?

En ORM (Object Relational Mapper) fungerer som en bro mellem to forskellige verdener:

- **Object-verdenen:** C#-klasser, objekter, properties og metoder.
- **Database-verdenen:** Tabeller, kolonner, primære nøgler og foreignkeys.

Klassediagram



Kommenterede [LH1]: Kan ikke se en skid af hvad det beskriver, næsten bedre at have det nede i bilag og så referer til det

Kommenterede [SH2R1]: Tilføj SvendeproeveRapporter/ClsassDiagram.png at main · stef663k/SvendeproeveRapporter til bilag og lav kort beskrivelse

Modeller/DTO'er og AutoMapper

Systemets datastyring er organiseret gennem en konsekvent struktur af modeller, DTO'er og mappere, der sikrer data kun bruges hvor de skal forbedre beskyttelse af brugerne.

Modeller

Modellerne repræsenterer de grundlæggende properties i vores system og giver et direkte billede af vores databasestruktur.

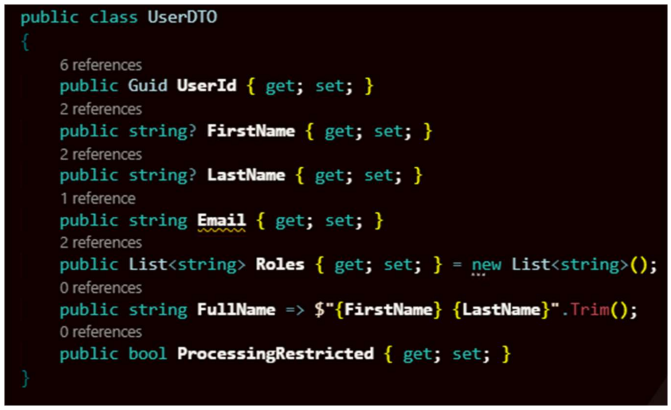
```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Threading.Tasks;

namespace SvendeApi.Models;

37 references
public class UserModel
{
    27 references
    public Guid UserId { get; set; }
    11 references
    public string? FirstName { get; set; }
    11 references
    public string? LastName { get; set; }
    [Required]
    [EmailAddress]
    30 references
    public string Email { get; set; }
    [Required]
    [MinLength(8)]
    [Compare("Password")]
    15 references
    public string Password { get; set; }
    14 references
    public bool IsActive { get; set; }
    20 references
    public Collection<UserRole>? UserRoles { get; set; }
    1 reference
    public Collection<PostModel>? Posts { get; set; }
    1 reference
    public Collection<CommentModel>? Comments { get; set; }
    1 reference
    public Collection<FollowerModel>? Followers { get; set; }
    1 reference
    public Collection<FollowerModel>? Followees { get; set; }
    1 reference
    public Colle
    1 reference
    public bool ProcessingRestricted { get; set; } = false;
}
```

bool UserModel.ProcessingRestricted { get; set; }

UserModel indeholder brugerens personlige data såsom loginoplysninger og referencer til roller og opslag. Modellen fungerer som de interne objekter der interagerer med Entity Framework og udgør grundlaget for alle databaseoperationer.

UserDTO (Data Transfer Object)

UserDTO fungere som den primære datakontrakt for brugerdata i API svar.



```
public class UserDTO
{
    6 references
    public Guid UserId { get; set; }
    2 references
    public string? FirstName { get; set; }
    2 references
    public string? LastName { get; set; }
    1 reference
    public string Email { get; set; }
    2 references
    public List<string> Roles { get; set; } = new List<string>();
    0 references
    public string FullName => $"{FirstName} {LastName}".Trim();
    0 references
    public bool ProcessingRestricted { get; set; }
}
```

DTO'en eksponerer kun de nødvendige brugeroplysniger uden at afsløre følsomme data som password eller interne database ID'er. Den inkluderer en beregnet FullName property der kombinerer for- og efternavn, samt en liste over rollenavne der transformeres fra UserRoles modellen som er en samlingstabel imellem bruger og roller. Dette sikrer dataminimering og en ren separation mellem modellerne og forretningslogikken.

AutoMapper

UserMapper profilen definerer de automatiske mappingregler mellem UserModel og UserDTO.

```
public UserMapper()
{
    // Mapping fra UserModel til UserDTO, konverterer UserRoles til en liste af role navne
    CreateMap<UserModel, UserDTO>()
        .ForMember(dest => dest.Roles, opt => opt.MapFrom(src =>
            src.UserRoles.Select(ur => ur.Role.RoleName).ToList()));

    // Mapping fra CreateUserDTO til UserModel, genererer nyt GUID og initialiserer tomme collections
    CreateMap<CreateUserDTO, UserModel>()
        .ForMember(dest => dest.UserId, opt => opt.MapFrom(src => Guid.NewGuid()))
        .ForMember(dest => dest.UserRoles, opt => opt.MapFrom(src => new Collection<UserRole>()));

    // Mapping fra UpdateUserDTO til UserModel, opdaterer kun hvis værdi ikke er null
    CreateMap<UpdateUserDTO, UserModel>()
        .ForAllMembers(opt => opt.Condition((src, dest, srcMember) => srcMember != null));

    // Mapping fra UserModel til AuthResponseDTO, ignorerer Token og ExpiresAt da disse genereres i servi
    CreateMap<UserModel, AuthResponseDTO>()
        .ForMember(dest => dest.Token, opt => opt.Ignore())
        .ForMember(dest => dest.ExpiresAt, opt => opt.Ignore())
        .ForMember(dest => dest.User, opt => opt.MapFrom(src => src));
}
```

Den starter med at udtrække rollenavnene fra UserRoles modellen som så præsenterer dem som en simpel liste af strenge. Denne transformation gør det nemmere at bruge og den gør det derfor hurtigere for API'en at skulle udtrække data. Profilen håndterer også oprettelsen af nye brugere ved automatisk at genere GUID og initialisere en collection af UserRoles modellen.

Manuel Mapping Extensions

Som supplement til AutoMapper er der lavet en klasse "UserMapperExtyension" med metoden ToUserDTO der giver direkte kontrol over konverteringsprocessen.

```
public static UserDTO ToUserDTO(this UserModel user)
{
    return new UserDTO
    {
        UserId = user.UserId,
        FirstName = user.FirstName,
        LastName = user.LastName,
        Email = user.Email,
        Roles = user.UserRoles?.Select(ur => ur.Role.RoleName).ToList() ?? new List<string>(),
    };
}
```

Extension metoden implementerer en direkte konverterings proces, hvor hver enkelt property i UserModel tilskrives den tilsvarende property i UserDTO. Denne direkte tilskrivning sikrer fuld kontrol over mapping logikken og håndterer potentielle null referencer sikkert gennem brug af null betinget tjek. Tilgangen giver mulighed for at

implementerer komplekse transformationer og specifikke forretningsregler uden at være afhængig af eksterne mappers automatiske processor.

Denne arkitektur sikrer en klar adskillelse mellem modeller og services, hvor AutoMapper håndterer de alminelige mapping regler, og manuelle extension finjusterer kontrol når det er nødvendigt.

AppDbContext central databasetilgang

Alle systemets modeller (User, Role, Post, Comment, osv.) er samlet i AppDbContext, som er arvet fra EF Core's DFbContext. Denne klasse fungerer som hovedindgangen til databasen og har 3 hovedopgaver:

1. **DbSet Proberties:** Hver model får sin egen DbSet<T>, som repræsenterer en database-tabel.

```
22 references
public DbSet<UserModel> Users { get; set; }
20 references
public DbSet<RoleModel> Roles { get; set; }
1 reference
public DbSet<UserRole> UserRoles { get; set; }
9 references
```

2. **Relationer og Constraints:** I OnModelCreating-metoden konfigureres alle relationer og begrænsinger.
3. **Database Connection:** Håndtere forbindelsen til databasen og oversætter LINQ-forespørgsler til SQL.

Hvorfor bruge EF Core

EF Core levere type-sikkerhed ved at lade compileren validere database-forespørgsler, hvilket fanger fejl under udviklingen i stedet for runtime. Reducerer mængden af den boilerplate SQL-kode og giver mulighed for at fokusere på forretningslogikken i services. Migrations-funktionaliteten hhåndterer ændringer i databasestrukteren, mens EF Core's holder øje med, hvad der er ændret, og sørger for at databasen kun bliver opdateret med de ændringer der har betydning for databasen . Dette betyder at den ikke spilder tid på at lave de samme opdateringer igen og igen.

```
// Konfiguration af PostModel tabellen
modelBuilder.Entity<PostModel>(entity =>
{
    entity.HasKey(e => e.PostId);
    entity.Property(e => e.Content).IsRequired();
    //Denne del her function fortæller at createdAt og updatedAt skal have default værdien GETUTCDATE() i sql server.
    entity.Property(e => e.CreatedAt).HasDefaultValueSql("GETUTCDATE()");
    entity.Property(e => e.UpdatedAt).HasDefaultValueSql("GETUTCDATE()");

    entity.HasOne(e => e.Author)
        .WithMany(e => e.Posts)
        .HasForeignKey(e => e.AuthorId)
        .OnDelete(DeleteBehavior.Restrict); // Forhindrer cascade delete problemer

    entity.HasOne(e => e.ParentPost)
        .WithMany()
        .HasForeignKey(e => e.ParentPostId)
        .IsRequired(false)
        .OnDelete(DeleteBehavior.Restrict); // Self-referencing for reposts (optional)
});
```

Her kan der ses et eksempel på hvordan man definerer Posts og Comments, og CreatedAt og UpdatedAt automatisk bliver sat til GETUTCDATE i SQL. Linjen *entity*.HasOne(*e* => *e*.Author) .WithMany(*e* => *e*.Posts).HasForeignKey(*e* =>*e*.AuthorId).OnDelete(DeleteBehavior.Restrict); betyder, at en post knytter sig til en bruger, en bruger kan lave mange post og vi blokere for kaskadeslet, så en bruger ikke kan slettes uden at vi også håndtere brugerens opslag først hvilket også vil hjælpe med at opfylde GDPR til ”right to be forgotten”.

entity.HasOne(*e* => *e*.ParentPost).WithMany().HasForeignKey(*e* => *e*.ParentPostId)

.IsRequired(false).OnDelete(DeleteBehavior.Restrict);

De her linjer viser selv-referencen ParentPostId er valgfri og restriktiv delete sikre at parent slettes og child-post stadig eksistere, kommentarmodellen er konfigureret på samme måde. UserRole har en sammensat nøgle (UserId, RoleId) for at realisere many-to-many mellem bruger og roller. Followermodellen er selv-refererende many-to-many, da en følger også kan følges. Likemodellen har et unikt index (UserId, PostId), så samme bruger kun kan like en post én gang. Tilsammen bidrager disse til mapning på databaseniveau. Når en service opretter eller henter data, bruger den AppDbContext og EF Core til at se efter ændringer eller opretter ny data så oversættes dette til SQL så det kan oprette forespørgslen i databasen. Men modeldata bliver aldrig sendt direkte til klienten, den bliver først konverteret til DTO’er via. Automapper.

JWT

JSON Web Tokens implementeres som vores primære autentificeringmekanisme til at sikre API-endpoints og real-time kommunikation.

Token generering

Systemet genererer access tokens baseret på brugerens identitet og roller. Tokens opbygges med standard claims inklusive bruger-ID, email og tildelte roller. Hver token forsynes med en udløbsdato baseret på konfigurerede udløbs indstillinger.

```
2 references
private (string token, DateTime expiresAt) GenerateAccessTokenForUser(UserModel user)
{
    var key = _config["Jwt:Key"];
    var issuer = _config["Jwt:Issuer"];
    var audience = _config["Jwt:Audience"];
    var minutes = int.TryParse(_config["Jwt:AccessTokenExpiresInMinutes"], out var m) ? m : 30;

    if (string.IsNullOrEmpty(key))
        throw new InvalidOperationException("Jwt:Key is not configured");

    var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(key));
    var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);

    var claims = new List<Claim>
    {
        new Claim(JwtRegisteredClaimNames.Sub, user.UserId.ToString()),
        new Claim(JwtRegisteredClaimNames.Email, user.Email),
        new Claim(ClaimTypes.NameIdentifier, user.UserId.ToString()),
        new Claim(ClaimTypes.Name, user.Email)
    };
    var roleNames = user.UserRoles?
        .Select(ur => ur.Role?.RoleName)
        .Where(r => !string.IsNullOrWhiteSpace(r))
        ?? Enumerable.Empty<string?>();
    foreach (var role in roleNames)
    {
        claims.Add(new Claim(ClaimTypes.Role, role!));
    }

    var expires = DateTime.UtcNow.AddMinutes(minutes);
    var token = new JwtSecurityToken(
        issuer: issuer,
        audience: audience,
        claims: claims,
        expires: expires,
        signingCredentials: credentials
    );
    var tokenHandler = new JwtSecurityTokenHandler().WriteToken(token);
    return (tokenHandler, expires);
}
```

Her ses metoden der generer tokens som har følgende hovedfunktioner:

1. Konfiguration hentes fra appsettings, denne bliver så overtaget af Azure App Service.
2. Sikkerhedsnøgle bliver oprettet ved hjælp af krypteringen HMAC-SHA256.

3. Claims bygges op med både standard JWT-claims og brugerdefinerede claims.
4. Roller tilføjes fra bruegerens USerRoles relation hvor der laves et null tjek som sikre at den ikke er null.
5. Token bliver nu genereret med alle parametre og returneres sammen med udløbstidspunkt.

Konfiguration og validering

JWT-middleware konfigureres til at validere tokens mod definerende parametre:

- JWT hemmelig nøgle fra konfiguration
- Issuer (API adresse) og audience (frontend) validering
- Levetidskontrol med konfigurerbar tidsrum for både access token og refresh token

Her ses et eksempel på hvordan denne konfiguration kan se ud:

```
"Jwt": {
  "Key": "kFj4bvQQ8GqxKectXPAB0g8edFQe9dQ8NM3gTJStIk2gbSOEbdVA/10wM9XNsBdf",
  "Issuer": "SvendeApi",
  "Audience": "SvendeApi",
  "AccessTokenExpiresInMinutes": 30,
  "RefreshTokenDays": 7
}
```

Denne konfiguration er kun blevet brugt i produktionen.

Sikkerhedsimplementering

Den hemmelige JWT-nøgle og de andre informationer fra konfigurationen er sikret igennem Azure App Service, API'en sikre dermed at hemmeligholde brugernes login information.

SignalR

SignalR implementere real-time tovejskommunikation I applikationen, hvilket muliggør øjeblikkelige opdateringer til tilkoblede klienter.

Hub konfiguration i Program.cs

```
builder.Services.AddSignalR();
```

Feedhub implementering

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.SignalR;

namespace SvendeApi.Hubs;

9 references
public class FeedHub : Hub
{
    0 references
    public override async Task OnConnectedAsync()
    {
        var userId = Context.User?.FindFirstValue(ClaimTypes.NameIdentifier);
        if (!string.IsNullOrEmpty(userId))
        {
            await Groups.AddToGroupAsync(Context.ConnectionId, $"user:{userId}");
        }
        await base.OnConnectedAsync();
    }
}
```

Ved oprettelse af forbindelse tilføjes brugeren til brugerspecifik gruppe på deres autentificerede UserId. Dette muliggør målrettede beskeder til specifikke brugere.

Integration i forretningslogikken

```
2 references
public async Task<bool> DeleteAsync(Guid commentId, Guid requestedUserId)
{
    var comment = await _context.Comments.FirstOrDefaultAsync(c => c.CommentId == commentId);
    if (comment == null)
        return false;
    if (comment.AuthorId != requestedUserId)
        return false;
    comment.IsDeleted = true;
    comment.UpdatedAt = DateTime.UtcNow;
    await _context.SaveChangesAsync();
    await _hubContext.Clients.All.SendAsync("CommentDeleted", commentId);
    return true;
}
```

I servicelaget bruges IHubContext til at broadcaste real-time events. Når en kommentar slettes, notificeres alle tilkoblede klienter øjeblikkeligt till CommentDeleted eventet.

Autentificering integrering

SignalR integreres med vores eksisterende JWT-autemtificeringssystem, hvilket sikre at:

- Kun autoriserede brugere kan etablere forbindelser

- Brugerens identitet hentes direkte fra JWT-tokenets claims
- Gruppetilmelding baseres på autentificerede UserId

Denne implantation sikrer at den er skalérbar, event-drevet kommunikationsplatform der integrerer nemt med eksisterende autentifikation og services.

Controller

I vores .NET Web API fungerer controllere som indgangspunkter for http anmodninger, hvor hver controller er specialiseret i at håndtere en specifik domænefunktionalitet. Controllere arver fra ControllerBase og specifikke attributter der definerer deres måde at fungere på.

```
[ApiController]
[Route("api/[controller]")]
[EnableCors("Default")]
[AllowAnonymous]
1 reference
public class AuthController : ControllerBase
{
```

AuthController er konfigureret med ApiController og Route attributter der fastlægger dens path til /api/auth, samt EnableCors for at tillade kommunikation med frontend applikationer.

API implementerer REST (Representational State Transfer) gennem et stateless design, hvilket betyder at serveren ikke gemmer information om brugerens session mellem hver forespørgsel. I stedet for at have en session på serveren der husker hvem brugeren er, skal klienten selv sende al nødvendig information med i hver eneste forespørgsel. Dette fungerer ved at klienten sender et JWT token i hver anmodning, som serveren kan validere uden at skulle slå op i en database.

Som følge af at API'en bruger REST-principper gør, at den konsekvent følger http-protokollen til CRUD-operationer. GET, POST, PUT og DELETE håndterer henholdsvis læsning, oprettelse, opdatering og sletning af data, mens endpoints som /api/Auth/login og /api/Auth/logout organiserer autentifikationsfunktionaliteten.

```
[HttpPost("login")]
0 references
public async Task<IActionResult> Login(LoginDTO loginDTO)
{
    try
    {
        var ipAddress = GetCLientIpAddress();
        var result = await _authService.LoginAsync(loginDTO, ipAddress);
        var cookieOptions = BuildRefreshCookieOptions(result.ExpiresAt);
        Response.Cookies.Append("refreshToken", result.RefreshToken, cookieOptions);
        return Ok(result);
    }
    catch (UnauthorizedAccessException ex)
    {
        return Unauthorized(new { message = ex.Message });
    }
    catch (Exception ex)
    {
        return StatusCode(500, new { message = "An error occurred while logging in", error = ex.Message });
    }
}
```

Login endpointed (/api/Auth/login) implementers somen HTTP POST anmodning, da denne metode er velegnet til at oprette en ny sessionsressource. Endpointed modtager brugerens email og password i en forespørgsels ”body” og returnerer en JWT token samt brugerdata ved succesfuld autentifikation. Ved fejl returneres passend http fejlkoder, som 401 Unautherozied ved ugyldige login data eller 400 ved manglende data.

```
[HttpPost("logout")]
0 references
public async Task<IActionResult> Logout(LogoutRequestDTO logoutRequestDTO)
{
    try
    {
        var ipAddress = GetCLientIpAddress();
        var token = logoutRequestDTO?.RefreshToken;
        if (string.IsNullOrWhiteSpace(token))
        {
            Request.Cookies.TryGetValue("refreshToken", out token);
        }
        if (string.IsNullOrWhiteSpace(token))
        {
            return Unauthorized(new { message = "No refresh token found" });
        }
        await _authService.LogoutAsync(token!, ipAddress);
        var cookieOptions = BuildRefreshCookieOptions(DateTime.UtcNow);
        Response.Cookies.Delete("refreshToken", cookieOptions);
        return Ok(new { message = "Logout successful" });
    }
    catch (UnauthorizedAccessException ex)
    {
        return Unauthorized(new { message = ex.Message });
    }
    catch (Exception ex)
    {
        return StatusCode(500, new { message = "An error occurred while logging out", error = ex.Message });
    }
}
```

Logout endpointed (/api/Auth/logout) implementers ligeledes som en HTTP POST anmodning, da kaldet involverer en ændring af sessionsressourcen. Endpointed gør at brugeren ikke kan sin refreshtoken da den bliver annulleret, og så fjerner den de relevante cookies, hvilket afslutter sessionen.

Services

Services i vores arkitektur fungerer som en forretningslogiklaget og er ansvarlige for at koordinere komplekse operationer, håndtere datastyring og implementerer domænelogikken. De fungerer som mellemled mellem controllere og modellerne, hvilket sikre en klar adskillelse af ansvarsområder og gør systemet mere skalérbar og nemmere at teste.

```
public async Task<AuthResponseDTO> LoginAsync(LoginDTO loginDTO, string? ipAddress = null)
{
    var email = loginDTO.Email.Trim().ToLowerInvariant();
    if (string.IsNullOrWhiteSpace(email))
        throw new UnauthorizedAccessException("Invalid email address");

    var user = await _context.Users
        .AsNoTracking()
        .Include(u => u.UserRoles!)
        .ThenInclude(ur => ur.Role)
        .FirstOrDefaultAsync(u => u.Email == email);

    if (user == null)
    {
        _logger.LogWarning("Invalid email address or password: {Email}", email);
        throw new UnauthorizedAccessException(_env.IsDevelopment() ? "User not found" : "Invalid credentials");
    }
    if (!PasswordHasher.VerifyPassword(loginDTO.Password, user.Password))
    {
        _logger.LogWarning("Login failed: password mismatch for user {UserId}", user.UserId);
        throw new UnauthorizedAccessException(_env.IsDevelopment() ? "Password mismatch" : "Invalid credentials");
    }
    if (!user.IsActive)
    {
        _logger.LogWarning("Login failed: user is not active: {UserId}", user.UserId);
        throw new UnauthorizedAccessException("User is deactivated");
    }

    var (token, expiresAt) = GenerateAccessTokenForUser(user);
    var (refreshToken, _) = await GenerateRefreshTokenForUserAsync(user, ipAddress);

    var response = _mapper.Map<AuthResponseDTO>(user);
    response.Token = token;
    response.RefreshToken = refreshToken;
    response.ExpiresAt = expiresAt;
    return response;
}
```

LoginAsync metoden i AuthService implementeret en komplet loginproces. Processen starter med validering og normalisering af input data, hvor e-mailadressen renses og konverteres til små bogstaver for at sikre en konsistent søgning i databasen.

Systemet henter herefter brugerens profil inklusiv deres tildelte roller fra databasen. Under dette trin anvendes AsNoTracking, som er en metode til at forbedre ydeevnen som fortæller Entity Framework, at den ikke skal overvåge ændringer til den hentede bruger. Dette reducerer hukommelsesforbruget, da systemet ikke forventer at opdatere brugerdata under loginprocessen, men kun læse dem.

Herefter gennemgår systemet en række sikkerhedstjek der inkluderer verifikation af brugerens eksistens, om der er indtastet det rigtige password og om kontoen er aktiv. Ved mislykket autentifikation logges detaljerede fejloplysninger til sikkerhedsovervågning, men der returneres generiske fejlmeddelser til klienten for at undgå at afsløre specifikke systemoplysninger som kunne misbruges.

Hvis autentifikationen lykkes, generes der en ny JWT access token med brugerens identitet og roller, samt et refreshtoken der tilbagekaldes når brugeren logger ud igen. Det endelige svar sammensættes gennem AutoMapper derfor transformerer brugerdata til det DTO'er, hvilket sikre at det kun er den nødvendige data også kaldet dataminimalisering og dermed minimere risikoen for datalækager.

Denne implementering består af flere lag af sikkerhed gennem passwordhashing, token baseret autentifikation, IP-sporing og fejlhåndtering.

Frontend i svelte

Applikations Arkitektur

Frontend applikationen er udviklet i Svelte, som er et moderne JavaScript framework der adskiller sig sig fra traditionelle frameworks ved at compile komponenter til almindelig JavaScript under buildtime. I modsætning til runtime baserede frameworks, Dette gør at Svelte er hurtig og ikke gør brug af en Virtual DOM (en kopi af UI).

Applikationen følger en SPA (Singe Page Achitecture) arkitekturen, hvor hele applikationen lever i en enkelt HTML fil, og navigationen mellem views håndteres klientside uden fuld side genindlæsning. Denne tilgang giver flydende brugeroplevelse der minder om mobil apps, med hurtige overgange og reduceret serverbelastning.

Login Komponent

```
async function handleLogin() {
            const response = await api(`/api/Auth/login`, {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ email, password })
    });
    console.log('login status: ', response.status);
    if (response.ok) {
        const data = await response.json().catch(() => null);
        if(data?.token) {
            setAccessToken(data.token);
        }
        goto('/dashboard');
    } else {
        console.error('login failed: ', await response.text().catch(() => ''));
    }
}
```

handleLogin funktionen i loginkomponenten, implementerer kernelogikken for bruger autentifikation. Funktionen initierer en http POST forespørgsel til ”/api/Auth/login” endpointed med brugerens e-mail og password. Ved modtagelse af et succesfuldt svar henter den JWT token ud fra response body og genmmer det via setAccessToken funktionen, hvilket sikrer at efterfølgende API kald inkluderer den nødvendige autorisation. Endelig håndteres navigation til dashboard ved brug af SvelteKits goto funktion, mens fejltilfælde logges til debugging.

Register Komponent

```
async function handleRegister() {
    const response = await api(`/api/User`, {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({
            firstName,
            lastName,
            email,
            password,
            confirmPassword,
            isActive: true,
            roles: ['User']
        })
    });
    if (response.ok) {
        goto('/login');
    } else {
        console.error('Register failed', await response.text().catch(() => ''));
    }
}
```

handleRegister funktionen administrerer brugeroprettelsen gennem en http POST forespørgsel til ”/api/User” endpointed. Funtionen konstruerer et request body der inlkuderer alle nødvendige brugeroplysninger samt standardværdier som ”isActive: true” og ”roles: {’User’}”. Ved succesfuld registrering omdirigeres brugeren til login siden, så brugeren kan logge ind med sin nye bruger. Og ved fejl sikre vi at API’en fejlkode bliver vist i konsollen.

Dashboard Komponent

Dashboard komponenten demonstrerer State management for realtidsopdateringer af posts- og kommentarer. Systemet bruger Sveltes reaktive programmering til at håndterer brugerinteraktioner og visuelle opdateringer i realtid.

submitComment funktionen demonstrerer håndtering af kommentaroprettelse.

```
async function submitComment(postId: string) {
    const text = (postIdToNewComment[postId] || '').trim();
    if (!text) return;
    expandedComments[postId] = true;
    if (!postIdToComments[postId]) postIdToComments[postId] = [];
    postIdToPosting[postId] = true;
    try {
        const res = await api('/api/Comment', {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({ postId, content: text })
        });
        if (res.ok) {
            postIdToNewComment[postId] = '';
            try {
                postIdToLoading[postId] = true;
                const refresh = await api(`/api/Comment?postId=${encodeURIComponent(postId)}&skip=0&take=50`);
                if (refresh.ok) {
                    postIdToComments[postId] = (await refresh.json().catch(() => postIdToComments[postId] || [])).map((c: any) => normalizeAuthor(c)) ?? (postIdToComments[postId] || [])
                    for (const c of postIdToComments[postId]) {
                        const uid = getUserIdFrom(c);
                        if (uid) void resolveUserName(uid);
                    }
                }
            } finally {
                postIdToLoading[postId] = false;
            }
        } else {
            console.error('Create comment failed', await res.text().catch(() => ''));
        }
    } finally {
        postIdToPosting[postId] = false;
    }
}
```

Funktionen starter med at validere inputteksten og derefter give øjeblikkelig visuel feedback ved at udvide kommentarsektionen og aktivere loading states. Den starter derfor et ny API anmodning til oprettelse af kommentaren og håndterer svaret ved at rydde inputfeltet og opdatere kommentarlisten for at sikre konsistens mellem grænseflade og data. Systemet administrer forskellige tilstande som postIdToPosting og postIdToLoading for at give brugeren tydelig feedback under processen, og implementerer fejlhåndtering der bevarer brugerens input ved fejl for at forbedre brugeroplevelsen. Dette sikrer at man får en responsiv interaktion hvor brugeren for umiddelbar bekræftelse på handlinger.

Layout Komponent

Layout komponenten ”theme system” implementerer tema håndtering og bevarer brugerens præferencer gennem localStorage og automatisk opdager systemet farve tema.

```
let theme = $state<'light' | 'dark'>('light');

function applyTheme(next: 'light' | 'dark') {
    const root = document.documentElement;
    if (next === 'dark') root.classList.add('dark');
    else root.classList.remove('dark');
}

function initTheme() {
    const stored = localStorage.getItem('theme');
    if (stored === 'light' || stored === 'dark') {
        theme = stored as any;
    } else {
        const prefersDark = window.matchMedia && window.matchMedia('(prefers-color-scheme: dark)').matches;
        theme = prefersDark ? 'dark' : 'light';
    }
    applyTheme(theme);
}

function toggleTheme() {
    theme = theme === 'dark' ? 'light' : 'dark';
    localStorage.setItem('theme', theme);
    applyTheme(theme);
}

onMount(() => {
    initTheme();
});
```

Ved komponent start kører initTheme funktionen som anvender enten tidligere gemt tema eller opdager systemets farve præferencer. toggleTheme funktionen sikre at når man skifter tema ved at den opdaterer DOM strukturen og sikre at tager temaet som er localStorafe, mens applyTheme funktionen administrer og implementerer temaet.

API laget

API laget integrerer token management der automatisk håndterer token der er udløbet. Systemet opdager 401 fejlen fra API’en og starter automatisk et kald til API ”/api/Auth/session” enpointed. Efter en succesfuld token fornyelse sendes den oprindelige forespørgsel uden at brugeren skal gøre noget, hvilket sikre en bedre brugeroplevelse, og sikre mindre fejl med autentifikation.

Test

I udviklingen af Agora systemet er der implementeret en teststrategi der sikrer kodens pålidelighed, vedligeholdsvenlighed og funktionelle korrekthed. Teststrategien omfatter unittests på modellaget, contract tests at API endpoint, og end-to-end tests af det komplette system. Denne tilgang kan sikre at fejl fanges tidligt så de kan opdages inden systemet er blevet for stort.

Unit Test

Unittests fokuserer på de enkelte komponenters funktionalitet.

```
ublic class PostModelUnitTests

    [Fact]
    0 references
    public void LavPost()
    {
        Console.WriteLine("Test bestået: Lav Post");

        // Arrange
        var authorId = Guid.NewGuid();
        Console.WriteLine($"AuthorId: {authorId}");

        // Act
        var post = new PostModel
        {
            PostId = Guid.NewGuid(),
            AuthorId = authorId,
            Content = "Dette er en test post",
            CreatedAt = DateTime.UtcNow,
            IsDeleted = false
        };
                        class System.String
        Console.WriteLine($"
                        Represents text as a sequence of UTF-16 code units.
        Console.WriteLine($"Content: {post.Content}");
        Console.WriteLine($"CreatedAt: {post.CreatedAt}");

        // Assert
        post.PostId.Should().NotBeEmpty();
        post.AuthorId.Should().Be(authorId);
        post.Content.Should().Be("Dette er en test post");
        post.IsDeleted.Should().BeFalse();

        Console.WriteLine("Test bestået: Post oprettet korrekt");
    }

    [Fact]
    0 references
    public void SletPost()
    {
        Console.WriteLine("Test bestået: Slet Post");

        // Arrange
        var post = new PostModel
        {
            PostId = Guid.NewGuid(),
            AuthorId = Guid.NewGuid(),
            Content = "Dette post skal slettes",
            IsDeleted = false
        };

        Console.WriteLine($"Post før sletning - PostId: {post.PostId}, IsDeleted: {post.IsDeleted}");

        // Act - Slet post ved at sætte IsDeleted til true
        post.IsDeleted = true;

        Console.WriteLine($"Post efter sletning - PostId: {post.PostId}, IsDeleted: {post.IsDeleted}");

        // Assert
        post.IsDeleted.Should().BeTrue();

        Console.WriteLine("Test bestået: Post slettet korrekt");
    }
```

I PostModelUnitTests klassen laves der to enkelte test på den properties. Først brugers der metoden LavPost til at oprette en post med alle de properties som den kræver. Bagefter laves der en SletPost for at slette den post kaldet lige har lavet og da kaldet anvender isDeleted flag laves der et blødt slet så den ikke bliver slettet. Dette er med til at sikre at systemet kan overholde GDPR krav om databevarelse.

UserModelUnitTests klassen implementerer tilsvarende tests for brugeradministration.

```
public class UserModelUnitTests
{
    [Fact]
    0 references
    public void OpretBruger()
    {
        Console.WriteLine("Test bestået: Opret Bruger");

        // Arrange & Act
        var user = new UserModel
        {
            UserId = Guid.NewGuid(),
            FirstName = "Test",
            LastName = "User",
            Email = "test@example.com",
            Password = "Password123!",
            IsActive = true
        };

        Console.WriteLine($"Bruger oprettet: {user.FirstName} {user.LastName}");
        Console.WriteLine($"Email: {user.Email}");
        Console.WriteLine($"UserId: {user.UserId}");

        // Assert
        user.UserId.Should().NotBeEmpty();
        user.FirstName.Should().Be("Test");
        user.LastName.Should().Be("User");
        user.Email.Should().Be("test@example.com");
        user.Password.Should().Be("Password123!");
        user.IsActive.Should().BeTrue();

        Console.WriteLine("Test bestået: Bruger oprettet korrekt");
    }

    [Fact]
    0 references
    public void SletBruger()
    {
        Console.WriteLine("Test bestået: Slet Bruger");

        // Arrange
        var user = new UserModel
        {
            UserId = Guid.NewGuid(),
            Email = "test@example.com",
            Password = "Password123!",
            IsActive = true
        };

        Console.WriteLine($"Bruger før sletning - Email: {user.Email}, IsActive: {user.IsActive}");

        // Act - Slet bruger ved at deaktivere og nulstille
        user.IsActive = false;
        user.Email = null;
        user.Password = null;

        Console.WriteLine($"Bruger efter sletning - Email: {user.Email}, IsActive: {user.IsActive}");

        // Assert
        user.IsActive.Should().BeFalse();
        user.Email.Should().BeNull();
        user.Password.Should().BeNull();

        Console.WriteLine("Test bestået: Bruger slettet korrekt");
    }
}
```

OpretBruger testen sikrer at brugerprofiler kan oprettes med alle nødvendige properties som personinfo, login oplysninger og aktivitetsstatus. SletBruger testen viser implementeringen af ”right to be forgotten” princippet gennem en deaktiveringsproces, hvor brugerens følsomme data nulstilles mens brugerens posts stadig vil være tilgængelige for at bevare dem til historiske referencer.

Testene bruger FluentAssertions biblioteket til at skrive tests som er nemme at få et overblik over, og det er nemt at se hvad ens forventninger og resultater vil være. Dette ses ved brugen af Arrange, Act og Assert som viser testen i et overskueligt omfang som kan ses i tesmetoderne hvor testdata opsættes, handlingerne udføres og resultaterne valideres i adskilte faser.

Contract Testing

Contract tests validerer at API'er overholder sine aftalte kontrakter med klienter ved at teste endpoints direkte igennem http anmodninger. Disse test verificerer både forespørgsler og svar, statuskoder, dataformater og fejlhåndtering for at sikre at API'et opfører sig som forventet.

Bruger test

Systemet har et testscenarie der validerer et komplet brugereksempel fra registrering til logout.

```
PS C:\Users\Stefan\Desktop\SvendePrøve\Api\SvendeApi> $Email="contract.tester+$(Get-Date -Format yyyyMMddHHmmss)@example
.com"; Invoke-RestMethod -Method POST -Uri 'https://agora-api-unique.azurewebsites.net/api/User' -ContentType 'applicati
on/json' -Body (@{email=$Email;password="StrongPwd!12345";confirmPassword="StrongPwd!12345";firstName="Contract";lastNam
e="Tester"}|ConvertTo-Json) -SessionVariable s


userId               : 79f1bfa0-2a98-4ca0-8cda-fa0bca9b6312
firstName            : Contract
lastName             : Tester
email                : contract.tester+20251110165510@example.com
roles                : {}
fullName             : Contract Tester
processingRestricted : False
```

Testen starter med brugeroprettelse, hvor en unik e-mail genereres for at undgå konflikter. Systemet verificerer at brugeren oprettes korrekt med alle påkrævede felter inklusive automatisk tildelte GUID og standardværdier for roller og processingRestricted.

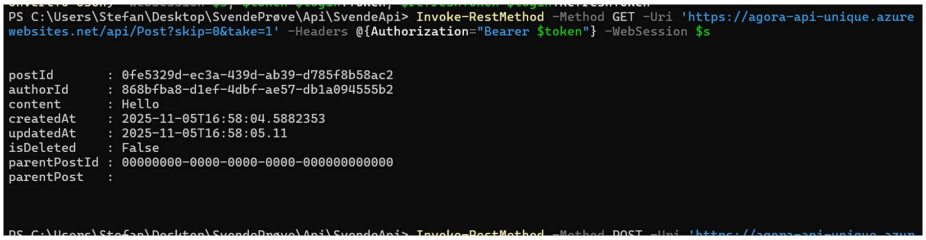
Efterfølgende testes login hvor systemet validerer at korrekte informationer accepterer, og returnerer både JWT access token og refresh token.

```
PS C:\Users\Stefan\Desktop\SvendePrøve\Api\SvendeApi> $login=Invoke-RestMethod -Method POST -Uri 'https://agora-api-uniq
ue.azurewebsites.net/api/Auth/login' -ContentType 'application/json' -Body (@{email=$Email;password="StrongPwd!12345"}|C
onvertTo-Json) -WebSession $s; $token=$login.Token; $refreshToken=$login.RefreshToken
```

Den succesfulde autentifikation demonstrerer systemets evne til at håndtere en sikker forbindelse.

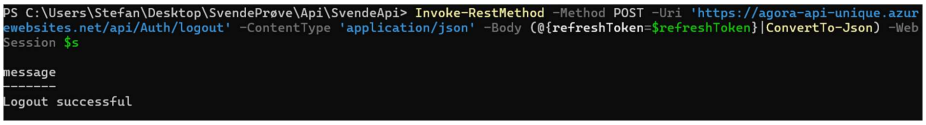
Data adgang og Logout

Med et gyldigt access token testes datahentning fra posts endpointed, hvilket verificerer både token baseret autorisation og korrekt dataformat.



Svar strukturen inkluderer alle forventede felter som postId, authorId, content, tidsstempler og referencer til parent post.

Afslutningsvis valideres logout processen hvor systemet bekræfter fjernelse af sessionen og returnerer en passende succesbesked.



Hele testen gennemføres med sessionen fortsætter via PowerShell's SessionVariable der simulerer en reel klients opførsel-

Disse contract tests sikrer at API'et opfylder sine forpligtelser overfor klienter.

Bruger Vejledning

Agora er et socialt medie, der er designet til at gøre det nemt at oprette opslag. Dele tanker og interagere med andre brugere gennem kommentarer og interaktioner som "likes". Systemet tilbyder en ren og intuitiv brugergrænseflade.

Installation

Webadgang

- Åben din browser og til https://agora-gray-six.vercel.app
- Systemet er optimeret til moderne browsere som Chrome.

Lokal installation

- Klon repository fra GitHub stef663k/SvendeFront: Frontend til svendeprøve eller git clone git@github.com:stef663k/SvendeFront.git
- Installer Node.js og afhængigheder via npm install
- Kør npm run dev for at kører den lokalt
- Backend API kører på https://agora-api-unique.azurewebsites.net/
- Klon repository fra GitHub stef663k/SvendeApi: Api til svendeprøve
- Ændre CORS i program.cs til hvad localhost din frontend åbner på
- Skriv dotnet ef database update
- Og skriv så dotnet run

Anvendelse

Registrering

1. Klik på ”Register ”som findes under ”Sign in”
2. Udfyld felterne: Fornavn, Efternavn, E-mail, Password
3. Bekræft dit password
4. Klik på ”Create account” for at oprette din konto

Login

1. Indtast din e-mail og password
2. Klik på ”Sign in” for at logge ind
3. Ved succes bliver du automatisk dirigeret til dit dashboard

Oprettelse af opslag

1. Skriv dit indhold i tekstfeltet ”Write something…”
2. Brug enter eller klik på ”Post” for at skrive din post
3. Dit opslag vil nu være synligt på siden

Oprettelse af kommentar

1. Klik på ”Comments” under et opslag for at se /oprette kommentarer
2. Skriv din kommentar i feltet
3. Klik på enter eller ”Send” for at tilføje en kommentar

Tema

- Klik på temaknappen øverst i højre hjørne for at skifte mellem lys og mørkt tema

- Systemet husker dit valg til næste gang

Data Export

- Klik på ”Export my data” for at downloade en kopi af dine oplysninger
- Filen downloades i tekstformat med alle dine opslag og kommentarer

Service

Fejlfinding

- Ved login problemer: Kontrollér e-mail og password
- Ved tekniske fejl: Prøv at genindlæse siden
- Sessioner udløber automatisk efter 30 minutters inaktivitet

Datahåndtering

- Du kan slette dine egne opslag og kommentarer via ”Delete” knappen
- Brug ”Export my data” for at se al din data
- Inaktive kontoer deaktiveres automatisk efter 6 måneder

Support

- Teknisk support tilbydes gennem systemets administrator
- Fejl og problemer kan rapporteres via https://github.com/stef663k/SvendeFront og opret et issue
- Systemet vil blive regelmæssigt opdateret