Høgskulen på Vestlandet

Fakultet for teknologi, miljø- og samfunnsvitenskap

Institutt for datateknologi, elektroteknologi og realfag

LinkUp - Plattformen som fremmer relasjoner

Systemdokumentasjon

Versjon 3.0

REVISJONSHISTORIE

Dato	Versjon	Beskrivelse	Forfatter
29/04/2024	1.0	Oppsett og igangsetting	Amund Fremming
06/05/2024	2.0	Fullføring og finpussing	Amund Fremming, Dennis Osmani og Adrian Berget
10/05/2024	3.0	Rettskriving og formatering	Dennis Osmani, Adrian Berget

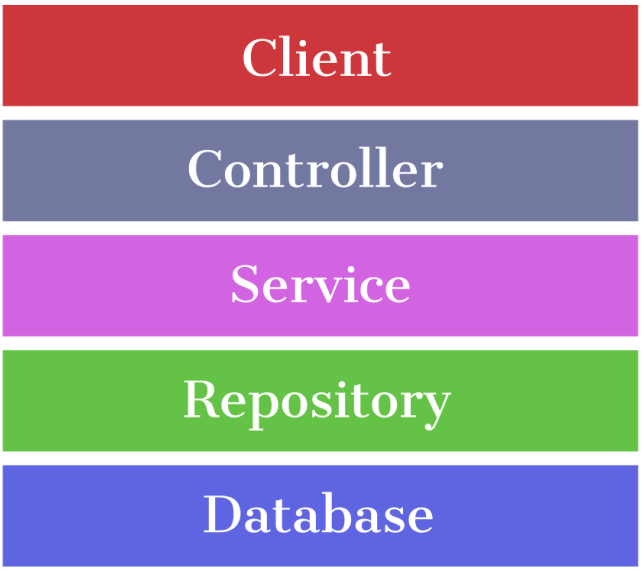
INNHOLDSFORTEGNELSE

1 INNLEDNING

Dette dokumentet er en del av støttedokumentasjonen for bachelorprosjektet, og har som hensikt å tilby en grundig og detaljert dokumentasjon av systemet som er utviklet, slik at lesere kan få en fullstendig forståelse av de tekniske og funksjonelle aspektene. Dokumentet fungerer som en veiledning for hvordan systemet er bygget og hvordan det opererer.

Dokumentet er en ressurs for utviklere, teknikere eller studenter som skal arbeide med kodebasen. Det gir nødvendige detaljer for vedlikehold og videreutvikling av systemet. Ved å følge denne dokumentasjonen sikres det at kodebasen kan operere effektivt og sikkert.

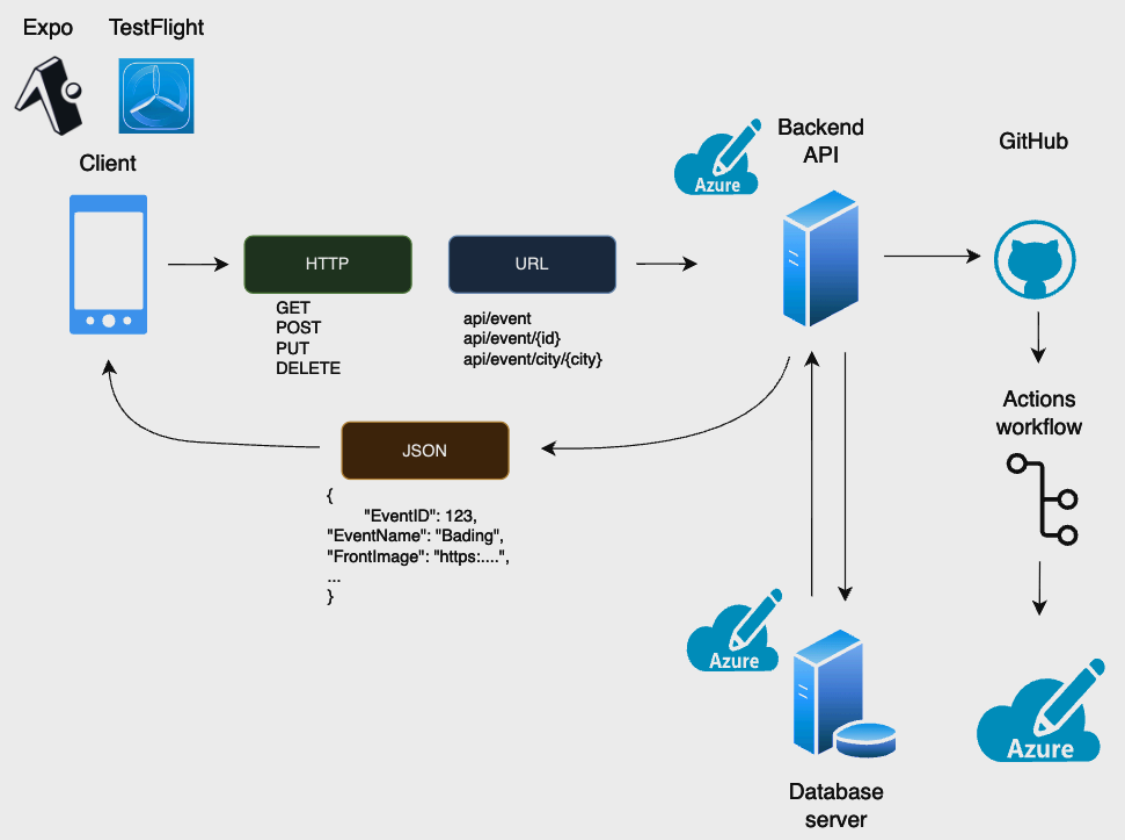
2 ARKITEKTUR



Figur 2.1 - Lagene til applikasjonen

Applikasjonen benytter en REST-arkitektur, som beskrives mer detaljert i kapittel 6. Her innehar hvert lag en distinkt rolle:

- **Klient:** Dette laget representerer det brukerne ser, altså klientgrensesnittet i applikasjonen. Det er koblet til backend-APIet gjennom URL-endepunkter ved bruk av JavaScripts Fetch-API.
- **Controller:** Dette laget definerer URL-endepunktene som klienten kobler seg til. Her fastsettes også hvilken HTTP-metode som skal returneres for de ulike forespørslene.
- **Service:** Inneholder applikasjonens forretningslogikk. Servicelaget håndterer unntak som genereres, og utfører nødvendige sjekker før det avgjør om en operasjon kan fortsette.
- **Repository:** Laget er ansvarlig for direkte operasjoner på data i databasen, og fungerer som et kommunikasjonsledd mellom database og forretningslogikk.
- **Database:** Reflekterer datamodellene som er opprettet, og har som hovedoppgave å oppbevare og lagre dataene til applikasjonen.



Figur 2.2 - Overordnet arkitektur

Figur 2.2 gir en helhetlig oversikt over systemets ulike komponenter. Verktøyet Expo benyttes for å simulere klientsiden under utvikling, mens TestFlight brukes for å teste applikasjonen i et miljø som etterlikner Apples App Store. Klientapplikasjonen kommuniserer med backend-APIet via HTTP-metoder som knyttes til en URL, hvor responsen og dataene formidles i JSON-format. Under utvikling kjøres backend lokalt på en datamaskin, og under testing med TestFlight kjøres det på Azure App Service, som er en serverløsning for å håndtere backend-koden.

Figuren illustrerer også forbindelsen til GitHub, som er lagringsplassen for gruppens kodebase. Den er integrert med GitHub Actions, som aktiveres når kode blir pushet til hovedgrenen (main branch). Dette utløser en bygge- og utrullingsprosess, hvor backend-koden automatisk implementeres på gruppens Azure App Service. I stedet for å etablere databasen selv, benyttes skolens databaseserver som er operativ på en separat Azure-server.

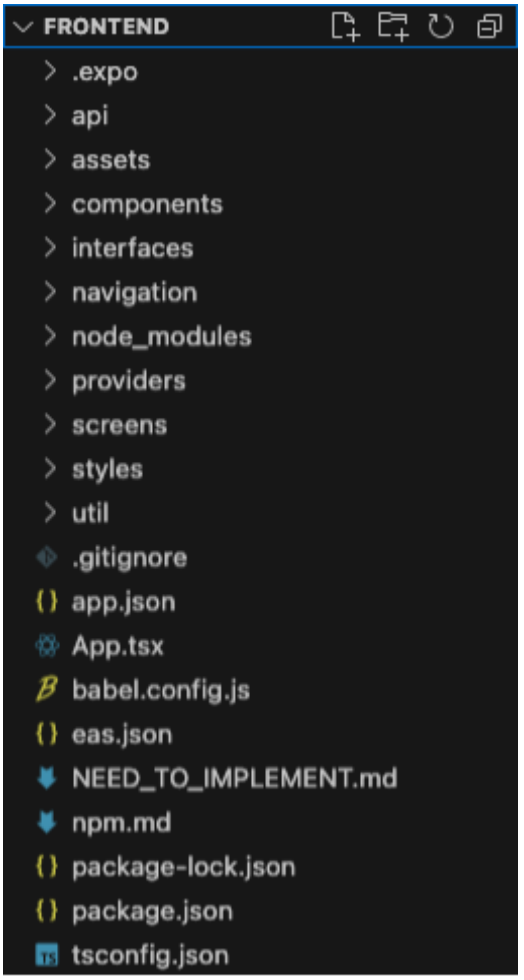
3 PROSJEKTSTRUKTUR

3.1 Frontend

3.1.1 Overordnet

Frontend-arkitekturen er i stor grad generert av Expos oppsettkommando. I tillegg til dette har gruppen opprettet flere mapper for å oppnå en bedre struktur, se figur 3.1. De ulike mappene inkluderer:

- **api:** Inneholder funksjoner for tilkobling til backend-APIet.
- **components:** Holder generiske komponenter som kan gjenbrukes flere steder i applikasjonen.
- **interfaces:** Definerer strukturer som samsvarer med dataene vi mottar fra og sender til backend.
- **navigation:** Administrerer tab-navigasjonen i applikasjonen.
- **providers:** Inneholder Context API-er for å håndtere global tilstand, slik som JWT-tokens.
- **screens:** Inneholder alle de forskjellige skjermene som vises i applikasjonen.
- **styles:** Definerer applikasjonens fargepaletter og skaleringsdimensjoner.
- **util:** Omfatter ulike hjelpeklasser (utility classes) som brukes på tvers av applikasjonen.

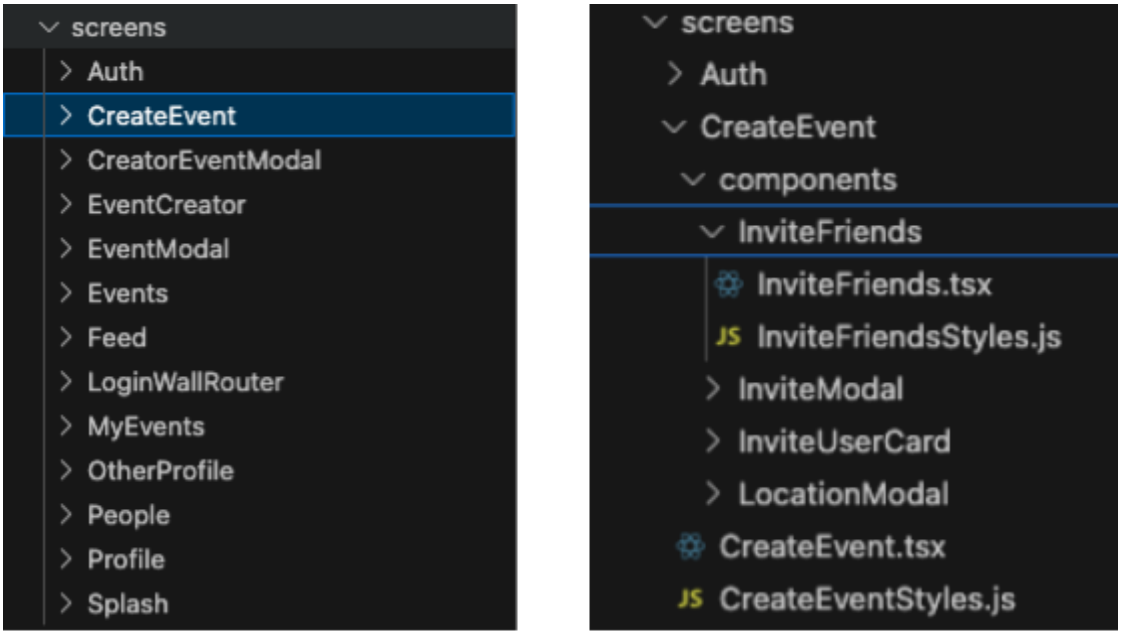


Figur 3.1 - Mappestrukturen til frontend

3.1.2 Screens

Mappen screens, se figur 3.2, representerer de visuelle skjermene i applikasjonen, hvor hver skjerm har sin egen dedikerte mappe. Noen av disse mappene inneholder kun to klasser: én for selve skjermkomponentens kode og én for tilhørende stildefinisjoner. Skjerm-mappene som inneholder mer funksjonsrike komponenter inkluderer en ytterligere undermappe kalt

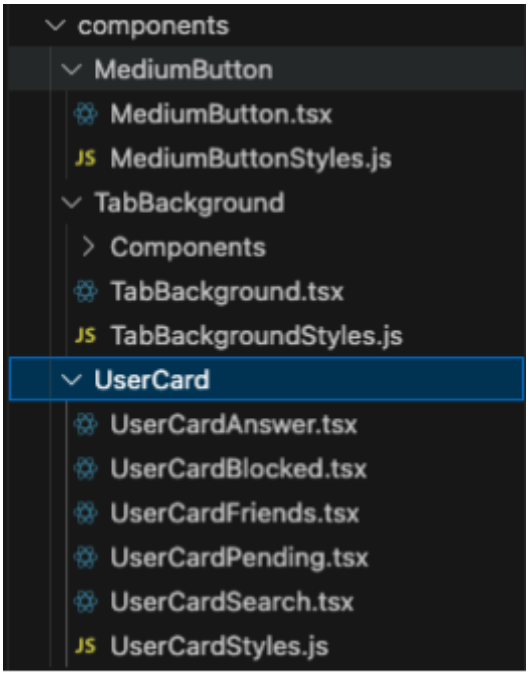
“components”. Disse komponentene er dedikerte komponenter for de ulike skjermene, og kan også inneholde flere under- mapper eller komponenter for en bedre oppdeling av funksjonaliteten for de ulike skjermene.



Figur 3.2 - Applikasjonens screens og strukturen til CreateEvent-screen

3.1.3 Generiske komponenter

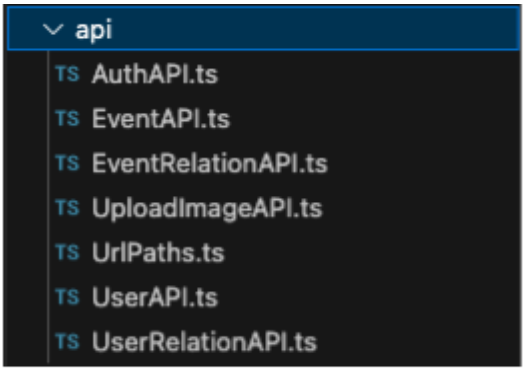
Det er også opprettet en mappe for generiske komponenter, som kjennetegnes ved deres fleksibilitet og mulighet til å bli brukt på tvers av flere deler av applikasjonen, uten å være bundet til en spesifikk skjerm eller komponent. Hver av disse generiske komponentene er organisert i sin egen mappe, som inneholder to filer: én for å definere det funksjonelle aspektet av komponentene og én for å definere tilhørende stil.



Figur 3.3 - Ulike komponenter

3.1.4 Api

Mappen API inneholder all nødvendig funksjonalitet for å etablere tilkoblinger til backend. Innenfor denne mappen er hver klasse designet for å korrespondere med en spesifikk kontrollmetode i backend. Videre har gruppen definert en fil kalt UrlPaths, som generaliserer URL-er til de ulike endepunktene. Dette har vist seg å være svært nyttig, spesielt når det har vært behov for å skifte mellom lokal kjøring og testing på produksjonsserver. Denne organiseringen sikrer effektiv håndtering av endepunkter og bidrar til en smidig overgang mellom ulike driftsmiljøer.



Figur 3.4 - API-ene som samhandler med backend

3.1.5 Package.json

"Package" filen, se figur 3.5, inneholder viktig informasjon relatert til prosjektets innstillinger og konfigurasjoner. Det mest sentrale aspektet ved denne filen er registreringen av alle avhengigheter (dependencies), som gruppen har benyttet seg av. En betydelig del av disse avhengighetene er relatert til navigasjon, hvor de bidrar til å muliggjøre et enkelt og raskt oppsett av tab-navigering. Andre avhengigheter fra Expo brukes for å integrere ferdiglagde brukergrensesnittkomponenter, som for eksempel "datetimepicker", som lar brukeren velge dato og tid på en enkel og brukervennlig måte.

Det er verdt å merke seg at flere av disse avhengighetene ikke lenger er i aktiv bruk i prosjektet. De er imidlertid ikke fjernet, da de kan være relevante for fremtidig utvikling av applikasjonen. Dette gjelder for eksempel bruken av Firebase.

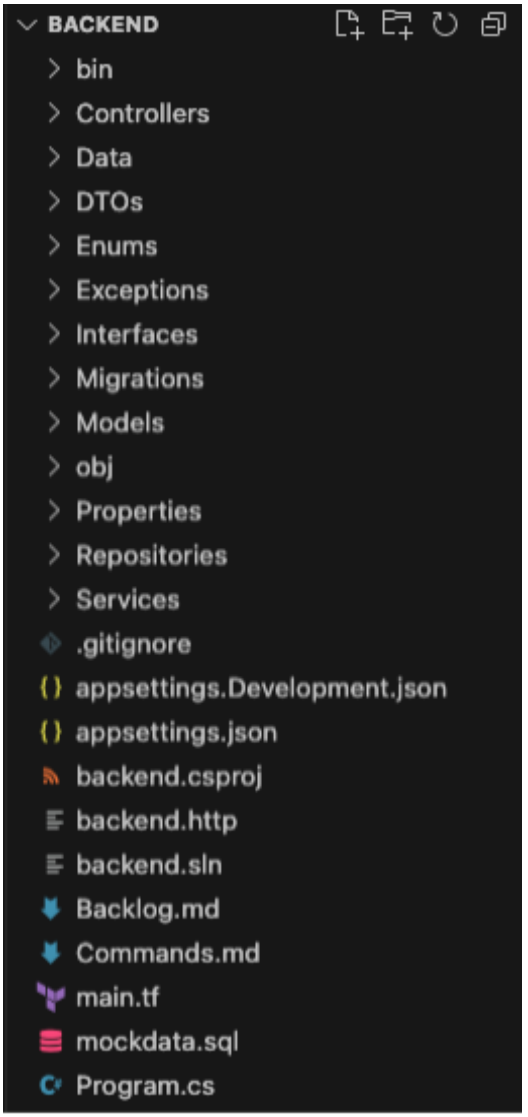
Figur 3.5 - Package.json filen med dependencies og konfigurasjoner

3.2 Backend

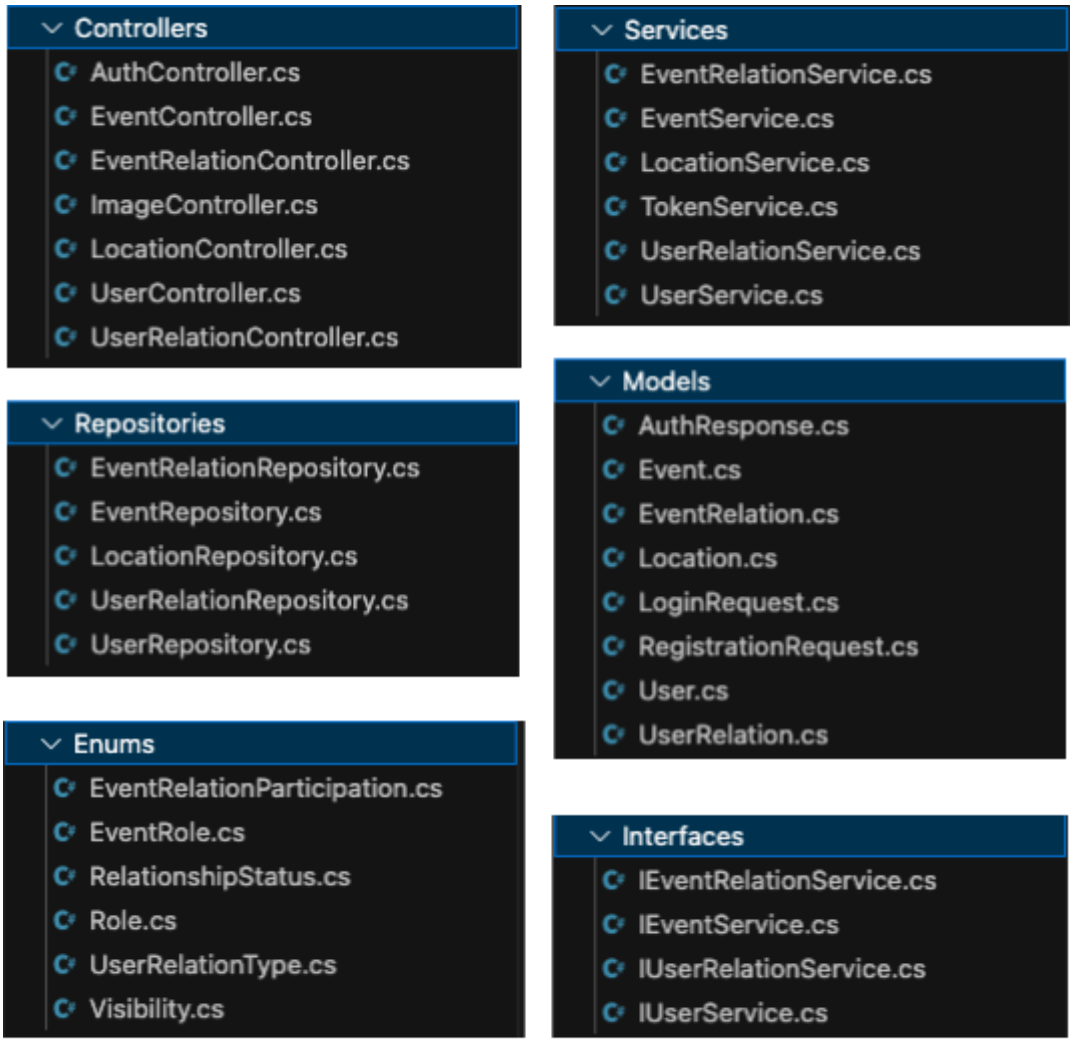
3.2.1 Overordnet

Mange av filene og mappene i backend-delen av prosjektet er generert av .NETs kommando for opprettelse av web API, men gruppen har også selv tilpasset arkitekturen til applikasjonens spesifikke behov, se figur 3.6. Strukturen inneholder mapper som reflekterer ulike aspekter av applikasjonen, beskrevet som følger:

- **Controllers:** Inneholder filer som håndterer brukerforespørsler og definerer responsen ved å samarbeide med Services og Repositories.
- **Services:** Holder forretningslogikken og fungerer som bindeledd mellom Controllere og Repositories.
- **Repositories:** Ansvarlig for direkte databasetilgang og dataoperasjoner, reflekterer operasjoner mot de ulike tabellene som er detaljert beskrevet i kapittel 5.
- **Models:** Definerer datastrukturer i koden som representerer databasetabellene.
- **Data:** Inneholder konfigurasjoner for Entity Framework Core, herunder definisjoner av relasjoner og databasens oppsett.
- **DTOs (Data Transfer Objects):** Holder forenklede objekter brukt for effektiv overføring av data mellom klient og server.
- **Enums:** Definerer enumerasjoner som brukes på tvers av applikasjonen for å standardisere koder og tilstander.
- **Interfaces:** Spesifiserer kontrakter som Services må implementere, sikrer konsistens og forutsigbarhet i metode implementeringene.
- **Migrations:** Inneholder migrasjonsfiler generert av Entity Framework Core for å håndtere databaseendringer over tid.



Figur 3.6 - Mappestrukturen til backend



Figur 3.7 - Ulike mapper og deres innhold

3.2.2 Swagger

I figur 3.7 nedenfor vises koden som konfigurerer Swagger, et verktøy brukt for å dokumentere og teste API-er for en backend-applikasjon (Swagger, u.å). Først aktiveres en API-utforsker, samt Swagger dokument-generering med “AddEndpointsApiExplorer()” og “AddSwaggerGen()”. I Swagger-konfigurasjonen spesifiseres også en ny dokumentversjon med tittelen "Backend API" og versjon "v1".

Videre legges det til en sikkerhetsdefinisjon for å støtte JWT (JSON Web Token) autorisasjon ved bruk av Bearer-skjemaet. Det defineres at autorisasjonstokenet skal inkluderes i HTTP-headeren med navnet "Authorization", og at bruk av token skal være i henhold til Bearer-skjemaet.

Til slutt angis et sikkerhetskrav som sørger for at alle API-endepunkter bruker denne sikkerhetskonfigurasjonen for autorisasjon, slik at APIet krever en gyldig JWT for tilgang. Dette sikrer at APIet er beskyttet mot uautorisert tilgang, og bidrar til å opprettholde applikasjonens sikkerhet.

Program.cs

```
// Swagger config
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new() { Title = "Backend API", Version = "v1" });

    // Define the OAuth2.0 scheme that's in use (i.e., Implicit Flow)
    c.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme
    {
        Description = "JWT Authorization header using the Bearer scheme. Example: \"Authorization: Bearer {token}\"",
        Name = "Authorization",
        In = ParameterLocation.Header,
        Type = SecuritySchemeType.ApiKey,
        Scheme = "Bearer"
    });

    c.AddSecurityRequirement(new OpenApiSecurityRequirement()
    {
        {
            new OpenApiSecurityScheme
            {
                Reference = new OpenApiReference
                {
                    Type = ReferenceType.SecurityScheme,
                    Id = "Bearer"
                },
                Scheme = "oauth2",
                Name = "Bearer",
                In = ParameterLocation.Header,
            },
            new List<string>()
        }
    });
});
```

Figur 3.7 - Oppsett av Swagger

3.2.3 Database og JWT

Koden nedenfor, se figur 3.8, viser hvordan applikasjonen skifter mellom konfigurasjoner for utviklings- og produksjonsmiljøet basert på en boolsk variabel, Azure Config. Dette blir nærmere forklart i delkapittel 8.3. Her blir det satt opp JWT-basert autentisering med sikkerhetsinnstillinger hentet fra Azure, for å sikre gyldige forespørsler. Ved “false” setter den opp tilsvarende konfigurasjoner lokalt ved hjelp av innstillinger fra appsettings.json, noe som gjør det mulig å veksle mellom kjøring lokalt og i skyen effektivt, og uten å endre kodebasen. Dette oppsettet bidrar til å forenkle utviklings- og testprosesser ved å tilpasse applikasjonens oppførsel basert på driftsmiljøet.

```
78  bool azureConfig = false;    // Endre denne til false for localhost kjøring!
79
80  if (azureConfig)
81  {
82      // Database setup with azure secrets
83      var databaseConnectionString = Environment.GetEnvironmentVariable("DATABASE_CONNECTION_STRING");
84      builder.Services.AddDbContext<Data.AppDbContext>(options => options.UseNpgsql(databaseConnectionString));
85
86      // Jwt setup with azure secrets
87      var jwtIssuer = Environment.GetEnvironmentVariable("JWT_ISSUER");
88      var jwtAudience = Environment.GetEnvironmentVariable("JWT_AUDIENCE");
89      var jwtKey = Environment.GetEnvironmentVariable("JWT_KEY");
90      builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
91          .AddJwtBearer(options =>
92          {
93              options.TokenValidationParameters = new TokenValidationParameters
94              {
95                  ValidateIssuer = true,
96                  ValidateAudience = true,
97                  ValidateLifetime = true,
98                  ValidateIssuerSigningKey = true,
99                  ValidIssuer = jwtIssuer,
100                 ValidAudience = jwtAudience,
101                 IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(jwtKey!)),
102                 ClockSkew = TimeSpan.Zero
103             };
104         });
105 }
```

```
106 else
107 {
108     // Database setup for localhost with config in appsettings
109     builder.Services.AddDbContext<Data.AppDbContext>(options => options.UseNpgsql(builder.Configuration.GetConnectionString("DefaultConnection")));
110
111     // Jwt setup for localhost with config in appsettings
112     builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
113         .AddJwtBearer(options =>
114         {
115             options.TokenValidationParameters = new TokenValidationParameters
116             {
117                 ValidateIssuer = true,
118                 ValidateAudience = true,
119                 ValidateLifetime = true,
120                 ValidateIssuerSigningKey = true,
121                 ValidIssuer = builder.Configuration["Jwt:Issuer"],
122                 ValidAudience = builder.Configuration["Jwt:Audience"],
123                 IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"])),
124                 ClockSkew = TimeSpan.Zero
125             };
126         });
127 }
```

Figur 3.8 - Konfigurasjon og oppsett av driftsmiljø

3.2.4 Controller, service og repository

Kodesnutten i figur 3.9 fra backend-delen av prosjektet demonstrerer konfigurasjonen av tjenester og kontrollere som del av oppsettet i applikasjonen. Først legges kontrollene til sammen med en JSON-konfigurasjon som hindrer sykliske referanser ved serialisering av objekter. Dette er viktig for å unngå feil i behandlingen av data med toveis-avhengige referanser mellom klasser. Deretter registreres flere tjenester (Service-klasser) og repositorier som Scoped avhengigheter, noe som betyr at en ny instans av hver klasse skapes per forespørsel (ByteHide, u.å.). Dette sikrer at tjenester som EventService, UserService og tilhørende repositorier er tilgjengelige der de behøves.

Videre konfigureres Entity Framework Core, Database-Kontekst, tjenester for token-håndtering og passord-hashing, som alle er essensielle for autentiseringen og sikker datahåndtering i applikasjonen.

```
49
50  // Controller and Service config
51  builder.Services.AddControllers().AddJsonOptions(options =>
52  {
53      options.JsonSerializerOptions.ReferenceHandler = ReferenceHandler.IgnoreCycles;
54  });
55
56  // Controller config
57  builder.Services.AddControllers();
58
59  // Service config
60  builder.Services.AddScoped<IEventRelationService, EventRelationService>();
61  builder.Services.AddScoped<IEventService, EventService>();
62  builder.Services.AddScoped<IUserRelationService, UserRelationService>();
63  builder.Services.AddScoped<IUserService, UserService>();
64  builder.Services.AddScoped<LocationService>();
65
66  // Repository config
67  builder.Services.AddScoped<EventRelationRepository>();
68  builder.Services.AddScoped<EventRepository>();
69  builder.Services.AddScoped<UserRelationRepository>();
70  builder.Services.AddScoped<UserRepository>();
71  builder.Services.AddScoped<LocationRepository>();
72
73  // EF Core DbContext fonfig
74  builder.Services.AddScoped<TokenService>();
75  builder.Services.AddScoped<IPasswordHasher<User>, PasswordHasher<User>>();
```

Figur 3.9 - Oppsett og konfigurasjon av Controllere, Service, Repository og DbContext

3.2.5 Bygging og oppstart

Figur 3.10 viser initialiseringen og konfigurasjonen av ASP.Net Core rammeverket for systemet ved å bygge applikasjonen og sette opp en pipeline for HTTP-forespørsler. Swagger og Swagger UI aktiveres også her for å definere at verktøyet skal brukes. Applikasjonen implementerer HTTPS-omdirigering for sikrere kommunikasjon. Det settes også opp ruting

```
132  var app = builder.Build();
133
134  // Configure the HTTP request pipeline.
135  if (app.Environment.IsDevelopment())
136  {
137      app.UseSwagger();
138      app.UseSwaggerUI();
139  }
140
141  app.UseHttpsRedirection();
142  app.UseRouting();
143  app.UseAuthorization();
144  app.MapControllers();
145
146  app.Run();
```

Figur 3.10 - Applikasjons bygging

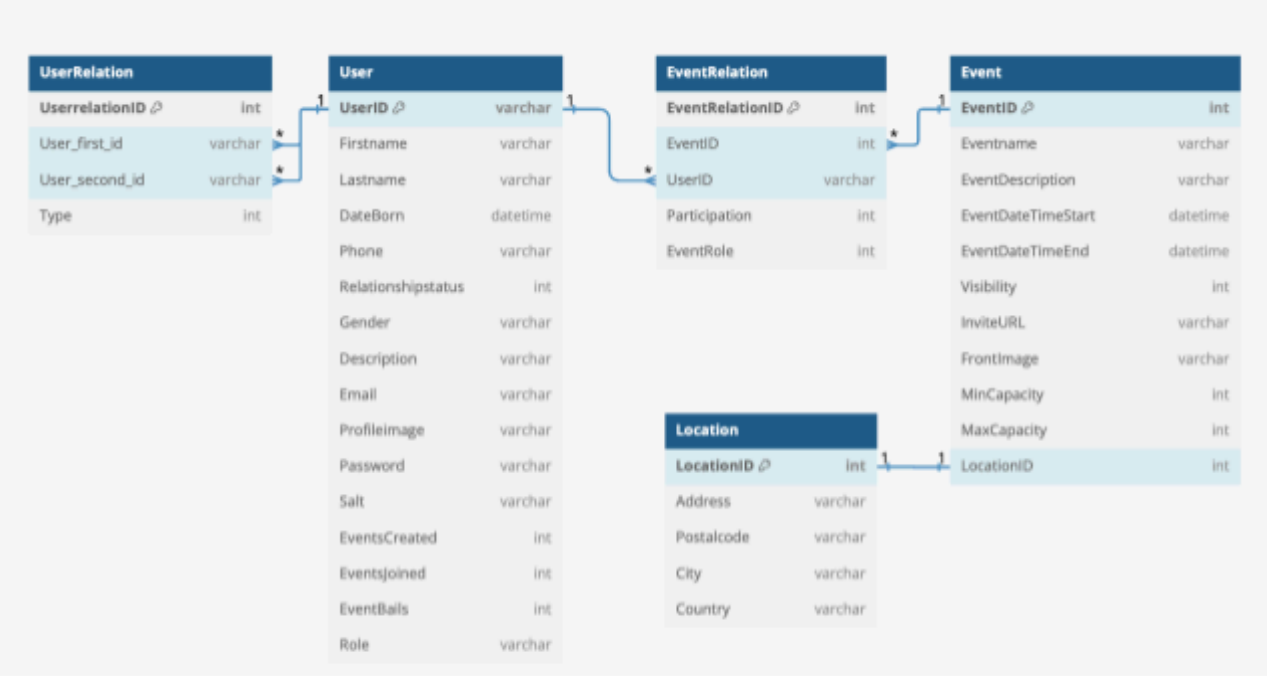
for håndtering av forespørsler basert på URL, samt autorisasjonskontroller. Controllere mappes til slutt for å koble forespørsler til riktige behandlingsmetoder, og deretter starter applikasjonen å lytte etter innkommende trafikk, og er dermed klar til å håndtere forespørsler.

3.2.5 Appsettings.json

JSON-konfigurasjonsfilen nedenfor, se figur 3.11, definerer forskjellige aspekter av en ASP.NET Core-applikasjon. Dette inkluderer blant annet bestemmelser for logging, tillatte verter, detaljer for PostgreSQL-database forbindelse, JWT-nøkler og identifikatorer, samt tilkobling detaljer for Azure Storage-kontoen. Disse innstillingene aktiveres bare i fravær av Azure-miljø.

Figur 3.11 - Appsettings.json

4 DATABASEMODELL



Figur 4.1 - ER-diagram av applikasjonen databasetabeller og deres relasjoner.

ER-diagrammet som vises i figur 4,1 er utviklet for å visualisere relasjonene mellom databasens tabeller. Primærnøkler fungerer som unike identifikatorer for hver rad, mens fremmednøkler refererer til primærnøkler i andre tabeller. Dette er med på å sikre integriteten mellom sammenkoblede data. For eksempel, i Event-tabellen er “EventID” primærnøkkelen og “LocationID” fremmednøkkelen som refererer til 'Location'-tabellen, som sikrer at hvert event er riktig knyttet til sin lokasjon.

Tabellene User, Event og Location representerer brukere, arrangementer og lokasjoner i systemet. For å håndtere mange-til-mange relasjoner mellom brukere og arrangementer, benyttes en koblingstabell kalt EventRelation. Denne tabellen gjør det mulig å legge til ytterligere informasjon om hver relasjon. På samme måte fungerer UserRelation for å administrere forholdet mellom to brukere, for eksempel i form av venneforespørsler og liknende.

5 SERVER-TJENESTER

REST (Representational State Transfer) er et arkitekturmønster, som brukes i utviklingen av applikasjonen. REST utnytter en stateless klient-server kommunikasjonsprotokoll som kan lagres i hurtigbufferen (Elkstein, u.å.). Protokollen er i de fleste tilfeller basert på HTTP. Hovedprinsippet i REST er manipulasjon av webressurser ved bruk av HTTP-verbene GET, POST, PUT og DELETE. Disse metodene korresponderer med følgende funksjoner: GET for datauthenting, POST for opprettelse av ressurser, PUT for oppdatering, og DELETE for sletting av ressurser.

I systemet har gruppen definert ulike REST-ressurser, der hver har sitt eget formål og ansvarsområde. Disse ressursene inkluderer:

- **Location:** Dette endepunktet håndterer operasjoner knyttet til lokasjoner, slik som opprettelse, oppdatering, og sletting av lokasjonsdata.
- **Event:** Her kan brukere utføre operasjoner relatert til eventer, for eksempel å opprette nye-, oppdatere eksisterende-, eller hente informasjon om eventer.
- **User:** Endepunkt som tillater brukere å administrere sine egne kontoer, inkludert registrering, innlogging, og endring av brukerdata.
- **EventRelation og UserRelation:** Disse ressursene håndterer relasjoner mellom eventer og brukere. For eksempel kan brukere bli knyttet til spesifikke eventer, eller få tilgang til eventer gjennom invitasjoner eller deltakelse i eventer.

Hver av disse REST-ressursene har egne URL-er og tilhørende kontrollmetoder i backend. Responsen fra disse endepunktene blir sendt tilbake til klienten i JSON-format, og gir en effektiv måte å samhandle med systemet på gjennom HTTP-forespørsler.

CI/CD-pipelines ble etablert via GitHub Actions for å automatisere opplastingen av kodebasen til Azure App Service, med formålet å hoste den på en Azure-server. Denne prosessen ble primært satt opp for å fasilitere brukertestingen gjennom TestFlight. Backend API-et blir hostet i en App Service, som er inkludert i gruppens ressursgruppe på Azure plattformen. I tillegg inneholder denne ressursgruppen en bildekontainer som muliggjør opplasting og lagring av bilder til applikasjonen

6 SIKKERHET

6.1 Sertifikater og kryptert kommunikasjon

Under utviklingen har gruppen benyttet seg av HTTP, kommunikasjon uten kryptering, grunnet enkelt oppsett og manglende behov for denne sikkerheten i det lokale utviklingsmiljøet. Da gruppen skulle gjennomføre brukertestingen, ble backend-kodebasen som nevnt tidligere hostet gjennom Azure, og HTTPS ble dermed brukt for kommunikasjon.

6.2 Passord

Gruppen har utviklet et registrering- og innloggingssystem med fokus på sikker lagring av passord. Systemet implementerer flere sikkerhetstiltak for å beskytte sensitive brukerdata effektivt. Et av tiltakene er såkalt "salting" av passord. Dette innebærer at et tilfeldig generert tekststreng blir lagt til i hvert brukerpassord før det hashes, slik at like passord får ulik hashverdi. "Salting" hjelper til med å beskytte mot "dictionary attacks", hvor en angriper bruker en forhåndslaget tabell med hashede passordverdier for å dekryptere passord raskt (Arias, 2019).

Videre blir det saltede passordet hashet ved hjelp av enveis kryptografisk hashing. Dette betyr at passordet konverteres til en hash-verdi som ikke kan reverseres (Arias, 2019). .Net-plattformen tilbyr innebygd funksjonalitet for hashing og validering av passord gjennom et verktøy kalt "PasswordHasher".

Det er imidlertid også viktig å nevne en svakhet i applikasjonens nåværende implementasjon. Grunnet begrenset tid og ressurser, lagres både passordhashene og saltene i samme databasetabell. Dette svekker sikkerheten i tilfeller hvor hele databasen blir lekket. Lagring av disse elementene separat ville være mer ideelt, enten i forskjellige tabeller eller i forskjellige databaser. Ved å separere dem vil sikkerheten forbedres ytterligere, ettersom risikoen for samtidig eksponering av både hash og salt reduseres.

6.3 Autorisering og Token

For autorisering og tilgangsstyring til API-endepunktene har gruppen implementert token-basert autentisering ved bruk av JWT (JSON Web Tokens). JWT er bredt anerkjent som en

industristandard for slik autentisering på grunn av sin robusthet og fleksibilitet (JWT, u.å). Når en bruker logger inn eller registrerer seg, genereres et JWT som deretter sendes til klienten. Dette tokenet lagres i en "context provider" på klientsiden, og denne fungerer som en global state-holder for hele klientsiden.

JWT-en sendes videre med hver forespørsel fra klienten til serveren som en "Bearer token" i autorisasjonsheaderen. Dette sikrer at hver forespørsel kan verifiseres og autoriseres korrekt basert på brukerens tilgangsnivå. Tokenet inneholder viktige brukerdata, inkludert brukeridentifikasjon og tildelte roller, hvor det skilles mellom “User” og “SuperAdmin”. Disse rollene brukes til å implementere rollebasert tilgangskontroll, hvor tilgangen til forskjellige endepunkter er basert på brukerens rolle.

I tillegg til autorisasjon, spiller JWT en kritisk rolle i sikkerhetsmekanismen ved at det verifiserer om brukeren har rettighetene som kreves for å manipulere spesifikke ressurser og funksjoner. Brukeridentifikasjonen trekkes ut for å se om brukeren skal ha tilgang til å manipulere den forespurte ressursen. Dette er avgjørende for å sikre at sensitiv informasjon og funksjoner er beskyttet mot uautorisert tilgang.

6.4 Cross-site Scripting og Sql-injection

Planen var opprinnelig å gjennomføre grundig validering av brukerdata både på klient- og serversiden for å sikre applikasjonens integritet og sikkerhet. Grunnet tidspresset som diskuteres i hovedrapporten, ble implementeringen av inputvalidering nedprioritert. Til tross for dette, klarte gruppen å implementere andre viktige sikkerhetstiltak for å beskytte mot visse typer sikkerhetstrusler.

Et viktig tiltak var bruk av HTML-escaping på serversiden. HTML escaping innebærer at spesifikke tegn i brukergenererte data konverteres til deres HTML-entiteter. Dette forhindrer at potensielt skadelig kode som kan være en del av brukerinput, blir tolket som HTML eller JavaScript og kjøres i brukerens nettleser. Tiltaket er effektivt for å beskytte mot cross-site scripting (XSS)-angrep, hvor angripere kan injisere skadelig skript i sider som vises til andre brukere (SecureBrain, u.å.). Det er viktig å merke at om gruppen hadde mer tilgjengelig tid, ville

det blitt lagt til attributter som regulære uttrykk og lengdebegrensninger på alle variabler i objekter. Det ville også vært naturlig å legge til Javascript-enkoding for bedre sikkerhet.

Systemet benytter seg av et rammeverk som støtter ORM (Object-Relational Mapping) mellom applikasjonens repository-lag og databasen. Bruk av ORM tilbyr flere sikkerhetsfordeler, særlig mot SQL-injeksjoner. ORM-rammeverket parametriserer automatisk SQL-spørringer, noe som betyr at det skaper et sikkerhetslag ved å separere brukerdata fra SQL-kode. Dette hindrer angripere i å manipulere spørringer ved å injisere skadelig SQL-kode, som ellers kunne tillatt uautorisert tilgang til-, endring av- eller sletting av data.

Til tross for at fullstendig inputvalidering ikke ble implementert, bidro disse tiltakene til å øke applikasjonens sikkerhet betraktelig. Gruppen anerkjenner viktigheten av omfattende inputvalidering, og planlegger å prioritere dette i eventuelle fremtidige oppdateringer for å sikre enda bedre beskyttelse mot et bredere spekter av sikkerhetstrusler.

7 INSTALLASJON OG KJØRING

7.1 Viktigste avhengigheter

Gruppen har implementert flere avhengigheter, både store og små, for å effektivisere utviklingen av applikasjonen. På klientsiden har mindre avhengigheter spilt en nøkkelrolle, særlig for navigering og integrering av ferdiglagde brukergrensesnitt-komponenter for dato- og tidinntasting.

Expo er en av de største og mest sentrale avhengighetene som ble benyttet. Dette er et rammeverk som inkluderer en simulator, og gjør det mulig å teste og simulere lokal kode direkte på utviklerens egen mobiltelefon gjennom deres Expo Go applikasjon. Expo App Service (EAS) er en annen kritisk komponent fra Expo, som forenkler prosessen med å bygge og distribuere applikasjonen. Denne tjenesten gir utviklere muligheten til å automatisk kompilere og distribuere applikasjoner for ulike plattformer, noe som gjør det raskere og enklere å få appen ut i brukernes hender.

EF Core (Entity Framework Core) er et ORM rammeverk som har vært avgjørende for rask og effektiv utvikling av databasen. Ved å abstrahere databasetilgangen, har EF Core muliggjort en mer strømlinjeformet datahåndtering, samtidig som det har redusert mengden manuell SQL-kode som var nødvendig. Dette har ikke bare fremskyndet utviklingsprosessen, men har også spilt en viktig rolle i å sikre applikasjonen mot SQL-injeksjoner. Dette gjøres ved at spørringene konstrueres sikkert, ved hjelp av rammeverket uten direkte string-konkatenasjon.

7.2 Installasjon

For å kunne kjøre prosjektet er det nødvendig at visse teknologier allerede er installert på den lokale maskinen. Disse inkluderer:

- Dotnet 8.x
- Node (siste versjon)

Laste ned og åpne prosjektet

Prosjektet kan initialiseres ved å først laste ned zip-filen i vedlegg 5 - Kildekode, og deretter pakke den ut og navigere inn i mappen.

Installasjon av avhengigheter

For å installere avhengighetene til frontend-delen, navigeres det først til frontend-mappen og deretter kjøres en installasjonskommando:

```
cd frontend
```

```
npm install
```

Det kan være nødvendig å godkjenne noen av installasjonene underveis.

Tilsvarende for backend, navigeres det ut av frontend-mappen og inn i backend-mappen, før nødvendige pakker installeres:

```
cd ../backend
```

```
dotnet restore
```

Kjøring av prosjektet i Expo Go appen

Slik prosjektet leveres, er det kun nødvendig å kjøre frontend delen av applikasjonen ettersom backenden er hostet på Azure. Dette ble gjort for å unngå en langt mer komplisert prosess hvor den lokale hosten fra backendents perspektiv må gjøres offentlig for at applikasjonen på telefonen skal kunne brukes.

For å starte frontend, åpne et nytt terminalvindu, naviger til frontend-mappen og kjør følgende kommando:

```
npx expo start
```

Expo Go-appen lastes ned på en mobiltelefon, hvor applikasjonen kan simuleres ved å scanne QR-koden som vises i terminalvinduet der koden kjøres. Dette har ikke blitt testet med en Android-telefon da gruppen ikke hadde tilgang til dette. Oppførselen skal være lik på disse enhetene, men det kan ikke bekreftes fra gruppens side.

Kjøring av prosjektet i XCode Simulator

Ettersom gruppens prosjekt er levert for å kjøre på Expo GO-appen, krever kjøring i XCode-simulatoren endringer i koden. Dette gjøres ved å navigere inn i backend-mappen og deretter inn i Program.cs filen. Her må azureConfig variabelen settes til false. I tillegg må

URL-basen i frontend endres. Dette oppnås ved å navigere inn i frontend, deretter “api” mappen og filen urlPaths.ts. Her endres kommentarene slik at localhost blir benyttet istedenfor Azure base.

Med installasjonene på plass, kan backend kjøres direkte i samme terminalvindu:

```
dotnet run
```

For å starte frontend, åpne et nytt terminalvindu, naviger til frontend-mappen og kjør følgende kommando:

```
npx expo start
```

På macOS kan iOS-simulatoren startes ved å trykke “i” i terminalvinduet hvor frontenden kjører, gitt at XCode er installert på maskinen.

Testing av applikasjonen

For å teste applikasjonen vil det være et alternativ å registrere en ny brukerkonto og benytte seg av denne. Hvis det er ønskelig å teste gjennom en bruker som allerede har forhåndsdefinert data, kan følgende informasjon bli brukt til innlogging:

- Mail: *linkup@mail.no*
- Passord: *linkup123*

8 DOKUMENTASJON AV KILDEKODE

8.1 API-dokumentasjon

Det ble implementert en Swagger-dokumentasjon for backend API-et, som ble gjort tilgjengelig gjennom Swagger UI på “http://localhost:5173/swagger/index.html”. Denne dokumentasjonen tillot gruppen å utforske, samt interaktivt teste forskjellige API-endepunkter direkte fra en nettleser. Dette bidro til en bedre forståelse av systemet for utviklergruppen selv. Dokumentasjonen dekker nøye autentiseringsprosesser og fremgangsmåter for autorisering via "Authorize" funksjonen. Dette er mest essensielt for handlinger som krever brukeridentifikasjon, som for eksempel brukerregistrering (POST /api/auth/register) og innlogging (POST /api/auth/login).

Videre inneholder dokumentasjonen detaljerte beskrivelser av API-kall relatert til hendelsesstyring, som opprettelse- (POST /api/event/create), henting- (GET /api/event/{eventId}), og sletting (DELETE /api/event/{eventId}) av eventer. Swagger presenterer også utformingen på de aktuelle datatypene og entitetene i de ulike forespørslene, noe som gjorde det enklere å integrere disse APIene effektivt i frontend-koden.

8.2 Service- og repository-dokumentasjon

Som det kommer frem tidligere i dokumentet er servicer og repositorier to kritiske komponenter i arkitekturen, med hver sin distinkte rolle og dokumentasjonsstrategi. Servicene holder hovedsakelig all forretningslogikken, og deres funksjonalitet er grundig dokumentert gjennom service-interfaces. Dette gir en klar og konsis beskrivelse av alle tjenester systemet tilbyr, inkludert metoder for å håndtere brukerinteraksjoner og dataflyt.

Repositoriene har en mindre detaljert dokumentasjon, ettersom de primært utfører enkle dataoperasjoner, hvor funksjonaliteten reflekteres tydelig i metodenes navn. Denne tilnærmingen understreker deres rolle i datamanipulasjon uten integrering av forretningslogikk, noe som bidrar til en tydelig separasjon av bekymringer i systemet.

9 KONTINUERLIG INTEGRASJON OG TESTING

9.1 Bakgrunn

Under utviklingsfasen ble det ikke implementert et system for kontinuerlig integrasjon. Kapittel 5 beskriver hvordan gruppen etablerte CI-pipelines da applikasjonen skulle gjøres tilgjengelig for brukertesting. Denne pipelinen ble også utformet for å støtte fremtidige versjoner av prosjektet.

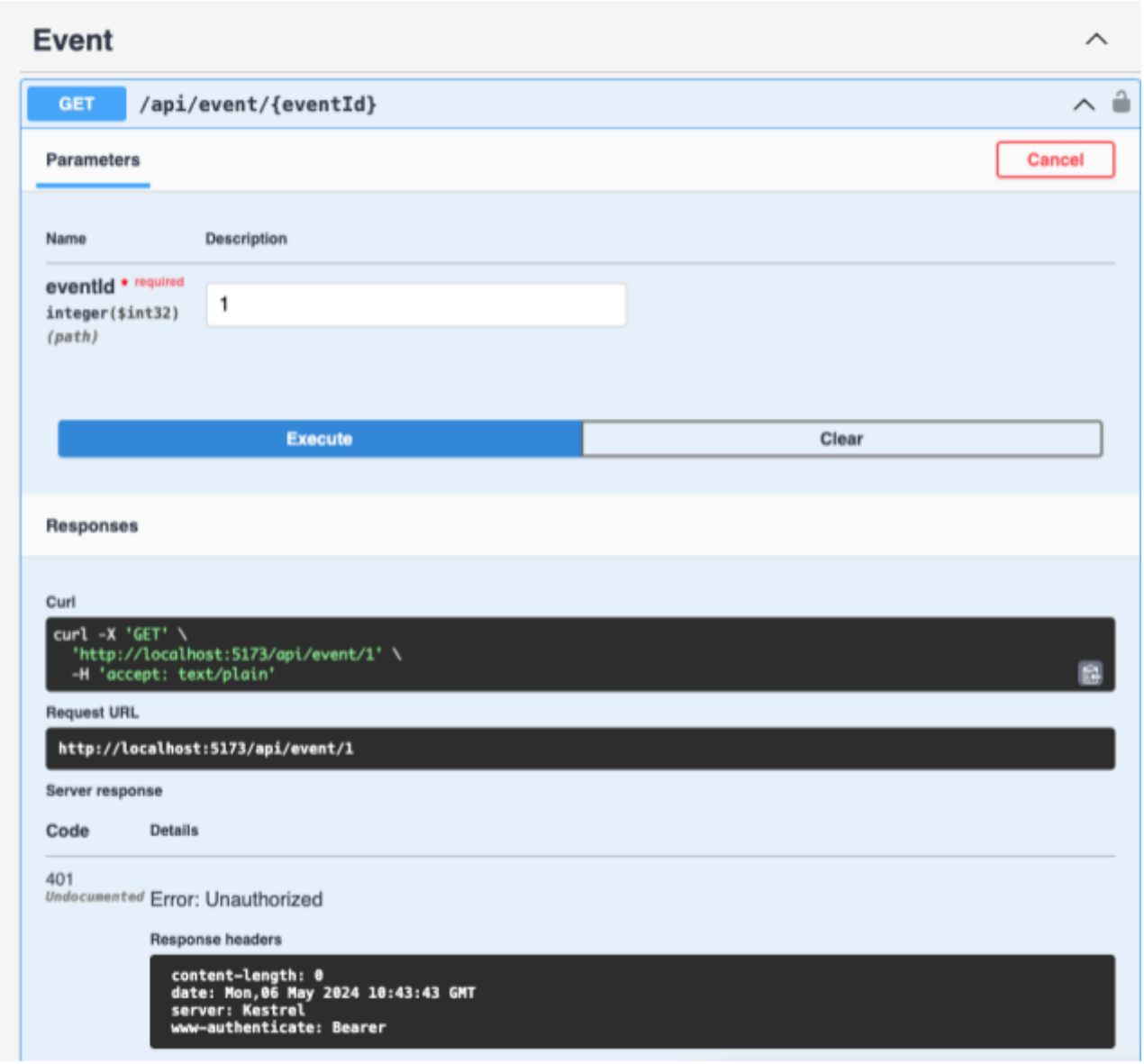
Det tidligere nevnte tidspresset i prosjektperioden førte til at det ble behov for å bortprioritere enkelte deler av systemet. For å kompensere for tapt tid og sikre levering av et komplett produkt, valgte gruppen å utelate automatiserte tester. Det er viktig å merke seg at automatiserte tester kunne ha vært raskere om testingen var dyp, men dette er noe gruppen hadde lite kjennskap til. Til tross for fraværet av automatiserte tester, var det essensielt å utføre manuelle tester for å minimere tidsforbruket på feilsøking.

9.2 Testing

Under utviklingen av backend-APIet, ble det utført manuelle tester på hvert endepunkt etter implementasjon. Denne prosessen innebar å kjøre backend lokalt og deretter åpne localhost på angitt port, etterfulgt av "/swagger". Her identifiserte gruppen det relevante endepunktet, og startet testingen uten innlogging for å verifisere at endepunktet var sikret. Deretter ble endepunktet testet for korrekt respons, samt at uautoriserte handlinger ble avvist. Det ble også kontrollert at korrekte HTTP-metoder, meldinger og feilmeldinger ble returnert. Gruppen gjennomførte to omfattende testrunder, hvor flere API-endepunkter ble testet samtidig for å sikre integrasjon og konsistens. I disse gjennomgangene ble hele backend-kodebasen grundig testet.

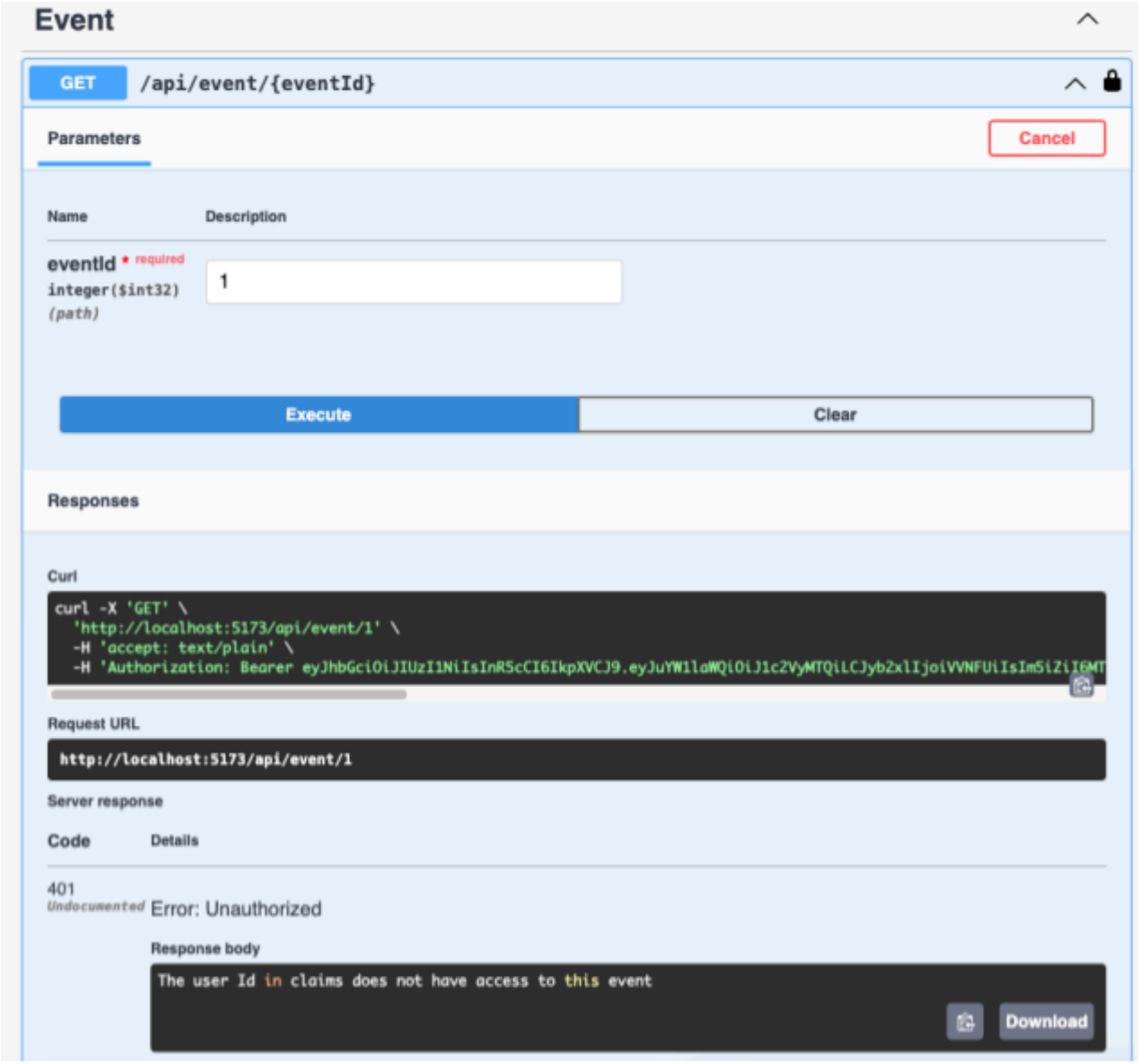
Gruppens fokus på å utvikle backend før frontend, kombinert med forsinkelser og tidspress, førte til at mindre tid ble viet til frontend-utviklingen, da denne fant sted mot slutten av prosjektperioden. I denne fasen utførte gruppen tester ved å koble frontend til backend for å verifisere funksjonalitet, presisjon og forventede resultater. Prosjektet ble avsluttet med en større brukertest og et brukerintervju, hvor det ble identifisert en rekke mindre feil, noen større problemer, samt områder for forbedring. Disse tilbakemeldingene ble integrert i de to siste sprintene av prosjektet, slik at gruppen fikk et mer solid og velfungerende sluttprodukt.

I figur 9.1 vises resultatet fra en manuell test i swagger der det forsøkes å hente informasjon om et event, når bruker ikke er logget inn og dermed ikke har et token i sin request.



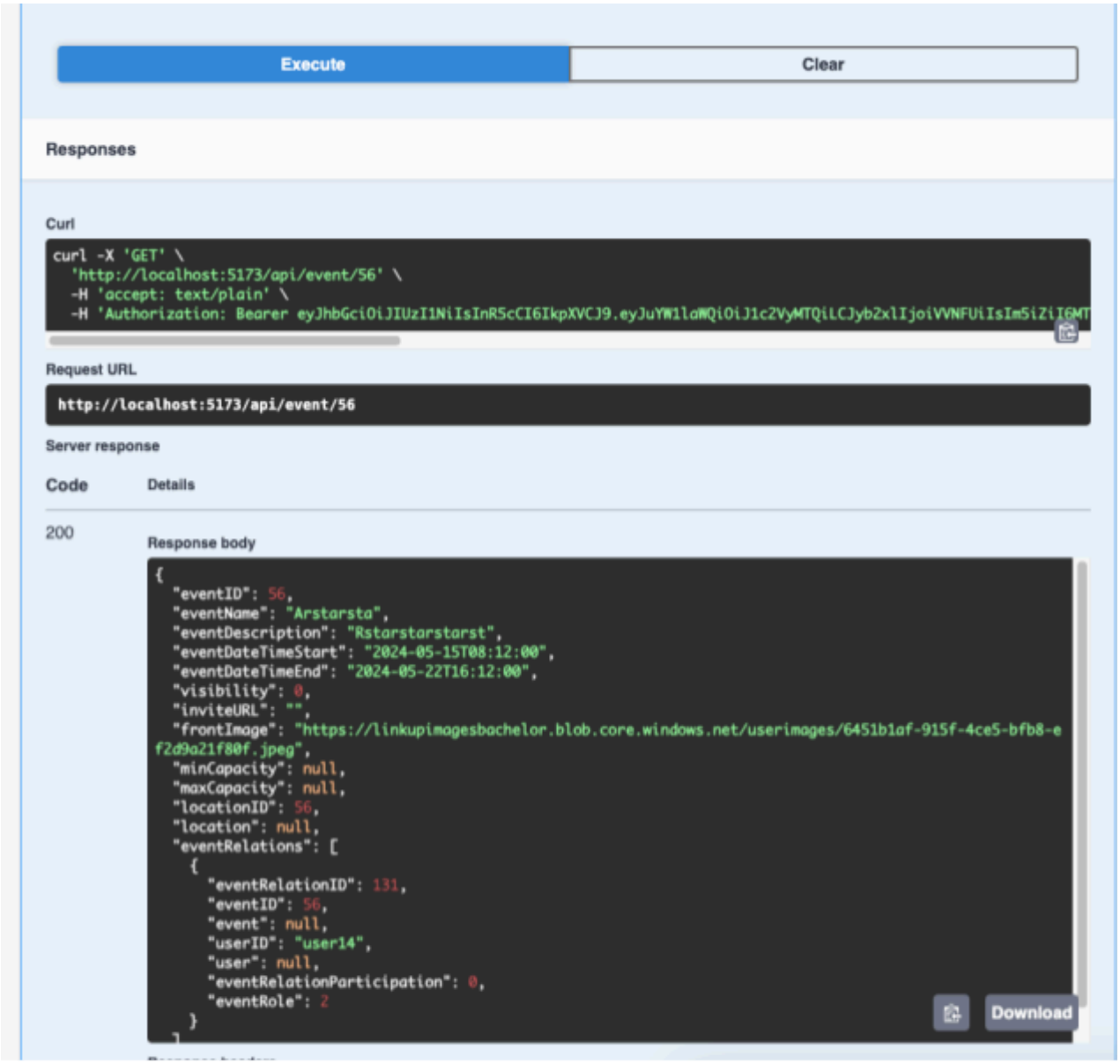
Figur 9.1 - Uautorisert request (Bruker har ikke et token)

I figur 9.2 vises resultatet av en bruker som forsøker å hente informasjon om et event hvor brukeren ikke har blitt med eller er deltaker, noe som ikke skal være mulig.



Figur 9.2 - Uautorisert request (Bruker har ikke tilgang)

I det siste eksempelet, se figur 9.3, er resultatet fra en forespørsel der brukeren forsøker å hente ut informasjonen til et event, hvor brukeren er innlogget, har et token med i requesten og er deltaker i eventet. Dette gir nødvendig tilgang til å se informasjonen om eventet.



Figur 9.3 - Autorisert request (Bruker har tilgang)

9.3 Potensielle forbedringer av tester

Dersom gruppen hadde mer tid til rådighet, ville implementering av automatiserte tester vært en prioritet for å forbedre og effektivisere utviklingsprosessen, samt sikre systemets funksjonalitet ytterligere. Automatiserte tester ville både redusert behovet for omfattende manuelle tester, samtidig som det ville øke tilliten til kodebasen ved kontinuerlig validering av funksjonalitet og sikkerhet ved hver endring.

Integrering av automatiserte tester tidlig i utviklingsfasen ville vært strategisk fordelaktig. Gruppen kunne oppdaget feil og problemer tidligere, som igjen ville redusert tidskostnadene knyttet til feilsøking og debugging i de senere stadiene av prosjektet. Ved å automatisere testing av grunnleggende funksjoner og kritiske integrasjonspunkter, kunne applikasjonen fremstått mer robust og pålitelig.

For eventuell videreutvikling av prosjektet, ville automatiserte tester ha tilbudt en stabil grunnmur som ny funksjonalitet kunne blitt basert på. For fremtidige utviklere ville det gitt høyere selvsikkerhet til å utføre endringer, ettersom automatiserte regresjonstester kunne validert at nye endringer ikke påvirker eksisterende funksjonalitet. Videre kunne implementering av en kontinuerlig integrasjons- og levering pipeline (CI/CD) med integrerte tester akselerert lanseringsprosessen for systemet.

10 REFERANSER

Arias, D (2019) *Hashing Passwords: One-Way Road to Security.* Tilgjengelig fra: https://auth0.com/blog/hashing-passwords-one-way-road-to-security. (Hentet 11.04.2024).

ByteHide (u.å.) *Scoped C#.* Tilgjengelig fra: https://www.bytehide.com/blog/scoped-csharp. (Hentet 06.05.2024).

Elkstein, D.M. (u.å.). *What is REST?* Tilgjengelig fra: http://rest.elkstein.org/2008/02/what-is-rest.html (Hentet 06.05.2024).

JWT (u.å.) *Introduction to JSON Web Tokens*. Tilgjengelig fra: https://jwt.io/introduction/. (Hentet 15.04.2024).

SecureBrain (u.å.) *XSS and SQL Injection Attacks*. Tilgjengelig fra: https://www.securebrain.co.jp/eng/blog/xss-and-sql-injection-attacks/. (Hentet 06.05.2024).

Swagger (u.å.) *What is Swagger?* Tilgjengelig fra: https://swagger.io/docs/specification/2-0/what-is-swagger. (Hentet 09.04.2024).