Høgskulen på Vestlandet

Fakultet for teknologi, miljø- og samfunnsvitenskap

Institutt for datateknologi, elektroteknologi og realfag

Mobilapplikasjon for automatisert måling av bevegelighet ved bruk av Kunstig Intelligens

Systemdokumentasjon

Versjon <1.3>

REVISJONSHISTORIE

Dato	Versjon	Beskrivelse	Forfatter
24.01.2024	1.0	Kapittel 2, Kapittel 4, Kapittel 5	Ole August Solem, Henrik Vallestad
17.04.2024	1.1	Lagt til nyeste versjon av arkitektur og filstruktur, skrevet tekst.	Ole August Solem, Henrik Vallestad
19.04.2024	1.2	Arkitektur, bilder	Ole August Solem, Henrik Vallestad
23.04.2024	1.3	Små endringer basert på tilbakemelding	Ole August Solem, Henrik Vallestad

INNHOLDSFORTEGNELSE

1 INNLEDNING

Dette dokumentet er skrevet for at leseren skal få en omfattende innsikt i systemet vi har utviklet. Det har som mål å gi en grundig forståelse av hva som er bakgrunnen for valgene som har blitt gjort, samt gi et detaljert overblikk over prosjektets struktur og arkitektur.

Ved å utforske systemets arkitektur og struktur, ønsker gruppen å bidra til å øke forståelsen for hvordan komponentene er organisert og samhandler. Videre er hensikten å tilby veiledning for implementeringen av systemet.

2 Arkitektur

2.1 Overordnet arkitektur

Den overordnede arkitekturen av applikasjonen illustreres i Figur 2-1, gir et helhetlig perspektiv på hvordan systemet er strukturer og hvordan komponentene samhandler.

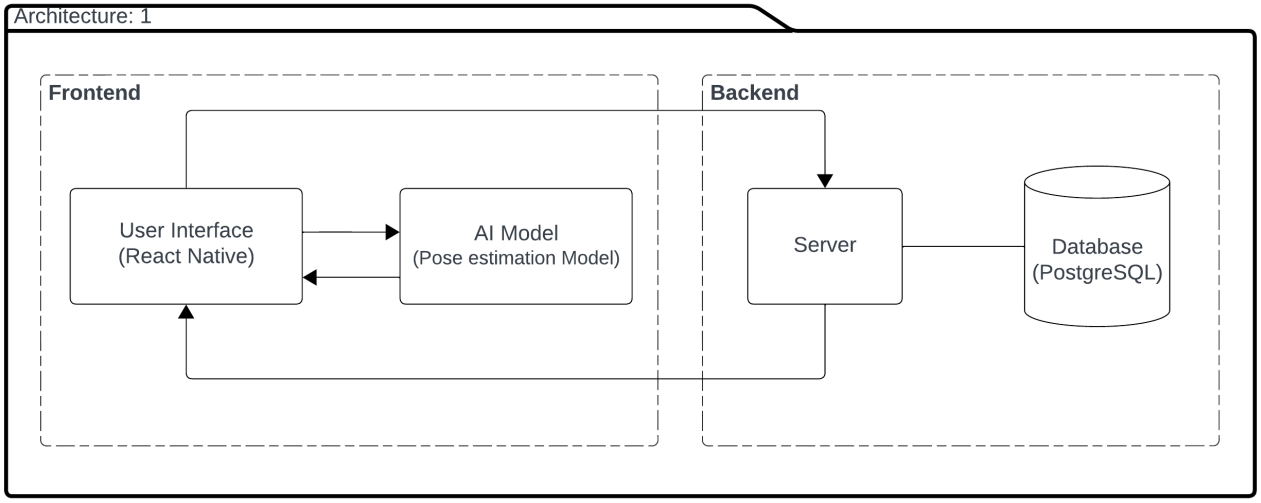
User Interface er mobilapplikasjonens brukergrensesnitt og er utviklet med React Native. Brukergrensesnittet kommuniserer med en AI model, som gir positur estimeringer.

Det er en kobling mellom brukergrensesnittet og tjenestene, implementert med ASP.NET.

Tjenestene fungerer som et bindeledd mellom mobilapplikasjonen og serveren, og er ansvarlige for å håndtere CRUD-operasjoner (Create, Read, Update, Delete) som brukerere utfører gjennom mobilapplikasjonen. Løsningen benytter HTTP for kommunikasjon mellom klient og server. Denne kommunikasjonsbroker sikrer effektiv dataoverføring mellom klienten og serveren.

Videre etableres en tilknytning mellom tjenestene og databasen, som er en PostgreSQL-database. Denne koblingen blir brukt til effektiv henting og lagring av data i systemet.

Denne arkitekturiske tilnærmingen legger vekt på klare kommunikasjonslinjer mellom systems ulike komponenter, og legger grunnlaget for en effektiv og skalerbar applikasjonsarkitektur.



Figur 2-1: Overordnet Arkitektur

2.2 Frontend

Frontend består primært av fire hovedkategorier av klasser: helpers, screens, components og AI model. Disse blir referert til som *hjelpere*, *skjermer*, *komponenter* og *KI modell*. Figur 2-2 viser arkitekturen for frontend (klientsiden).

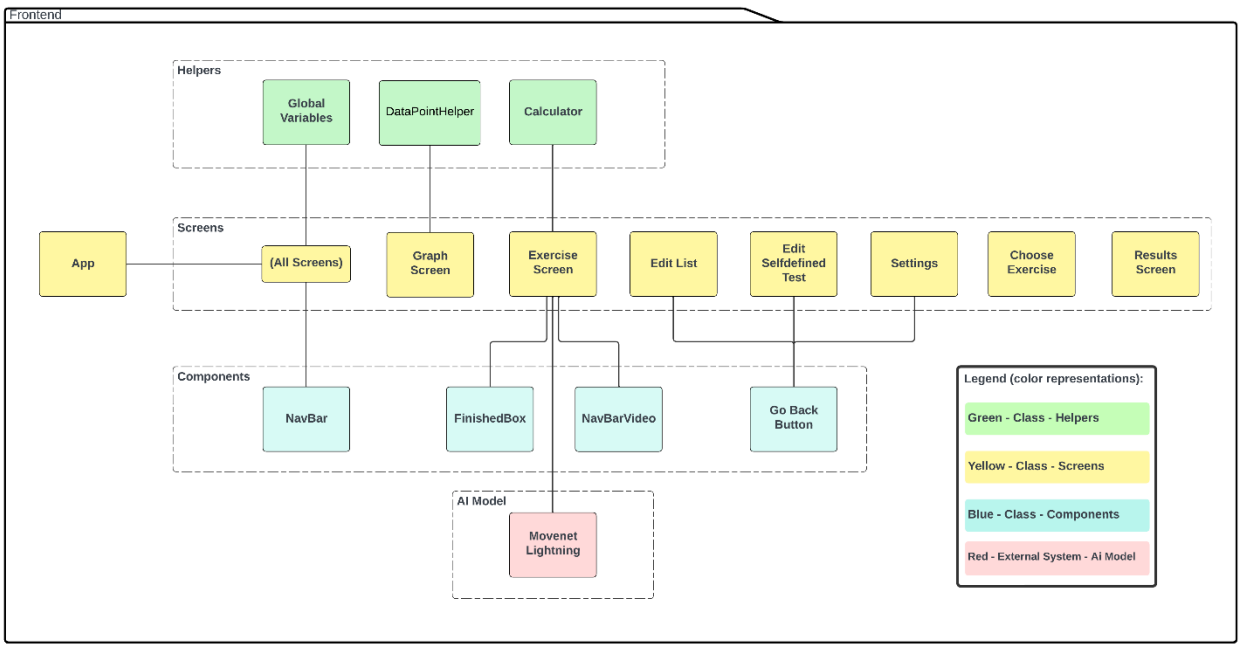
Appikasjonen starter med selve programmet som heter App. App importerer alle skjermer, og setter opp en navigasjonsstack mellom dem, slik at brukeren kan navigere mellom forskjellige deler av applikasjonen. Skjermene representerer de visuelle grensesnittene som brukeren ser, for eksempel Graf-skjermen *Graph Screen* eller Innstillinger *Settings*. Skjermene igjen importerer komponenter og hjelpere.

Komponenter er grafiske element som utfører samme funksjonalitet uavhengig av hvilken skjerm som importerer dem. Ved å importere og gjenbruke komponenter blir koden vår mer lesbar, skalerbar og endringsvennlig.

Hjelpere er filer som inneholder globale variabler og logikk for håndtering av funksjoner slik at skjermene skal være mer leselige og utvidbare.

En felles egenskap for alle skjermene er at de har en navbar og globale variables (globalvariables). Navigasjonslinjen er plassert nederst på skjermen og lar brukeren navigere mellom de viktigste skjermene. Globalvariables-filen inneholder kritisk informasjon som nødvendig for alle skjermer, for eksempel informasjon om brukeren og tilkoblingen til backend.

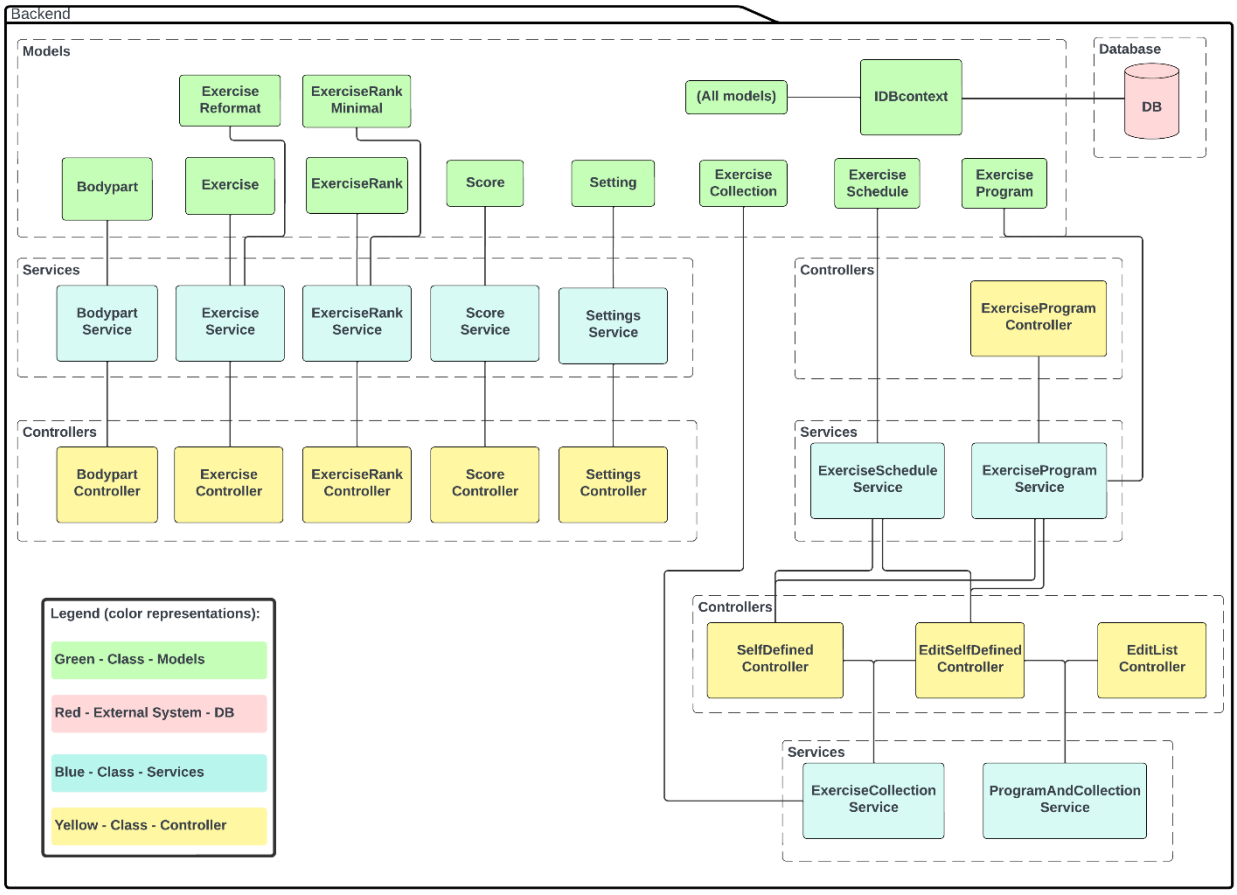
Denne strukturerte tilnærmingen bidrar til å organisere utviklingen av frontend-delen av systemet, og legger til rette for en effektiv og oversiktlig implementering.



Figur 2-2: Frontend Arkitektur

2.3 Backend

Backend systemet er delt opp i tre kategorier av klasser, Controller, Services og Models. Figur 2-3 viser arkitekturen for backend (serversiden). Systemet er bygget på en RESTfull struktur og benytter HTTP for kommunikasjon mellom klient og server. Kontroller kategorien er en overordnet samling av klasser, den håndterer kommunikasjon mellom frontend og backend ved bruk av HTTP forespørsler. Service kategorien har hovedansvar for kommunikasjon til databasen, Systemet benytter Entity framework (EF) for oppkobling til Azure databasen. Models kategorien har hovedansvar for definisjon av objekter. Kategorien benyttes av andre klasser for å strukturere data. Objektene i Models samsvarer med de korresponderende tabellene i Azure databasen. Løsningen benytter EFs dependency injection for sammenkobling av Objekter og kommunikasjon med databasen. Kall til backend systemet starter i hovedsak fra Controller klassen etter kall fra klientsiden hos brukeren. Controller klassen gjør så kall til Servicene som benytter AppContext for kommunikasjon med databasen. Objekter konstrueres så basert på klassene i Models.

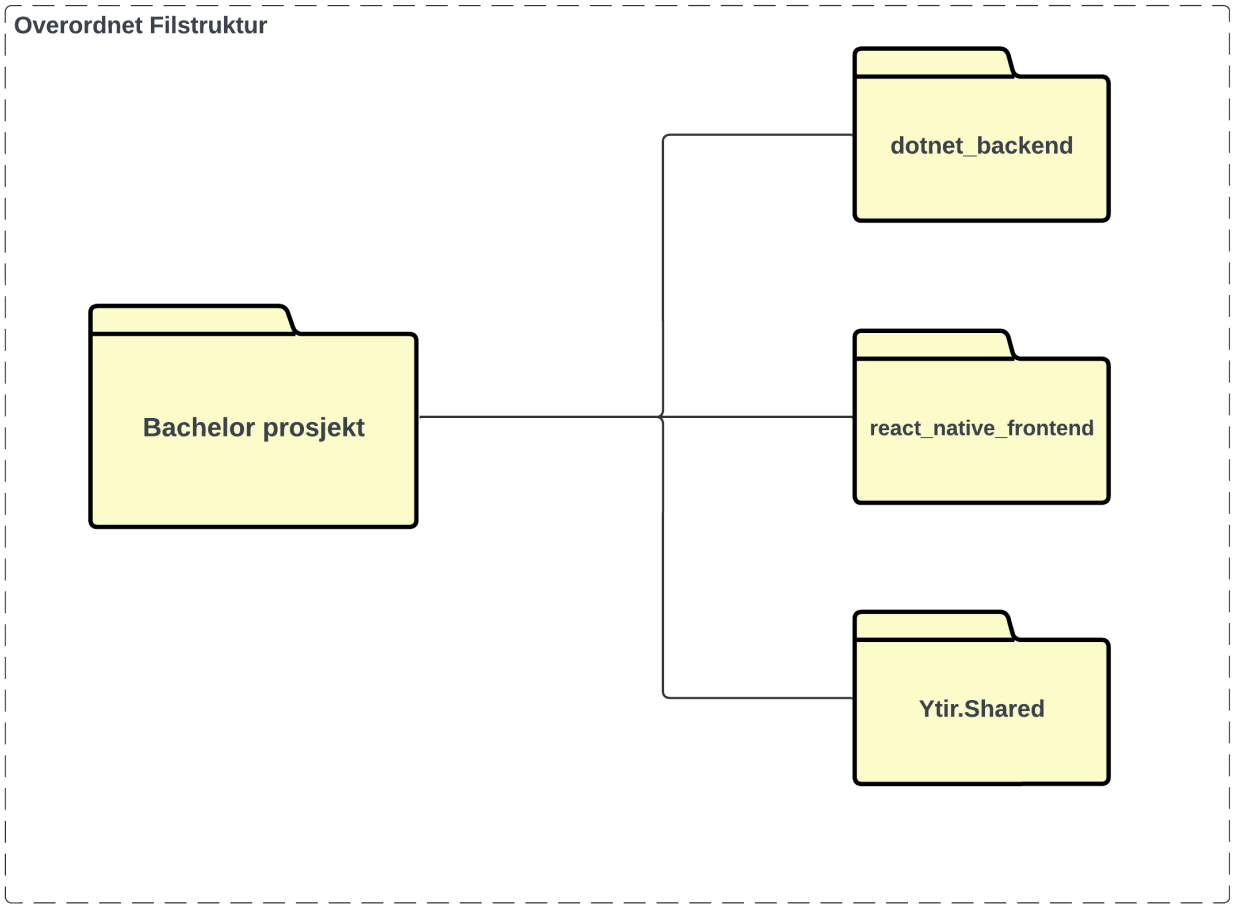


Figur 2-3: Backend Arkitektur

3 PROSJEKTSTRUKTUR

3.1 Overordnet filstruktur

Systemet er organisert som en tredelt hierarkisk struktur. Figur 3-1, viser filstrukturen for det overordnede systemet. Dotnet_backend og react_native_frontend er to separate miljø som kan deployes hver for seg.

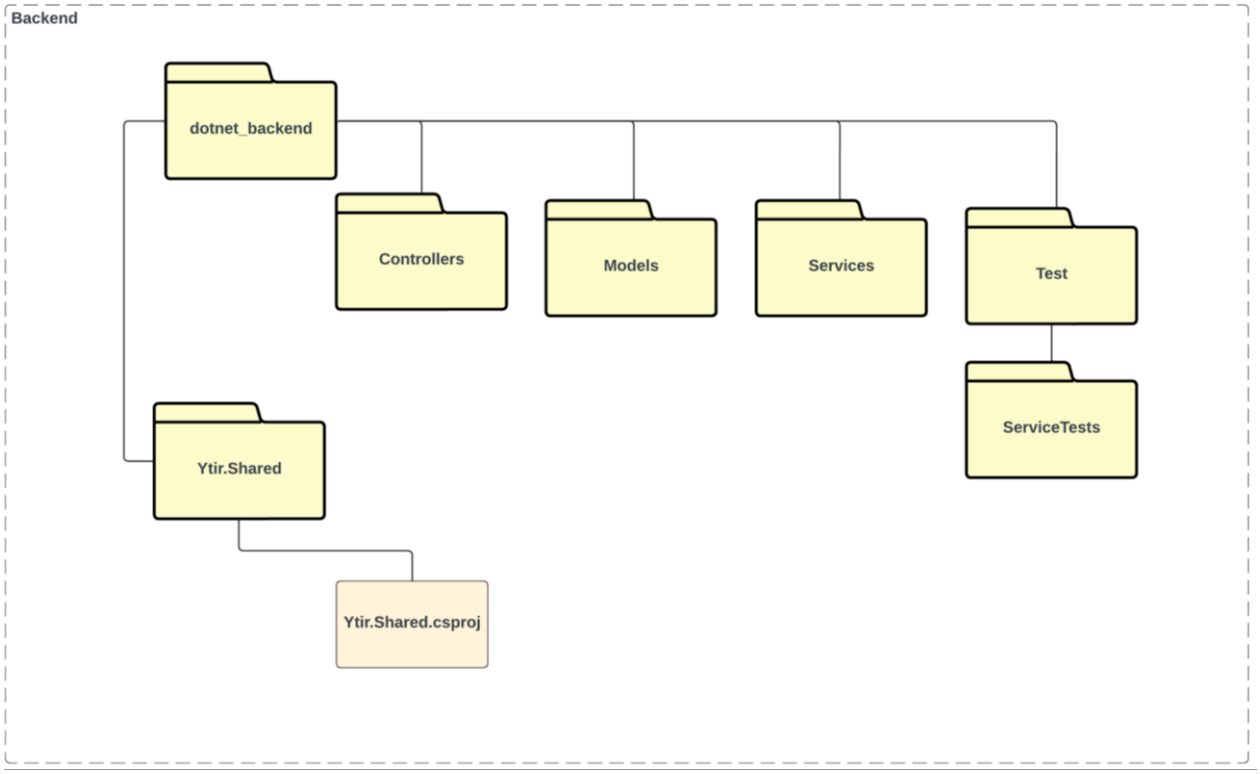


Figur 3-1: Filstruktur

3.2 Backend filstruktur

Figur 3-2 viser filstrukturen for serversiden. dotnet_backend inneholder kode for backend systemet og kommunikasjon mellom Azure databasen. Backend strukturen er delt opp i Controllers, Models, Services og Test mappene. Controllers har ansvar for HTTP forespørsler og bruk av Service klassene. Models inneholder logikken for instansiering av objekter. Service grenen er ansvarlig for kommunikasjon mellom backend systemet og databasen.

Ytir.Shared er en mappe som inneholder de felles avhengighetene mellom test miljøet og dotnet_backend miljøet. Både dotnet_backend og test importerer alle avhengighetene fra Ytir.Shared, noe reduserer duplikat kode, og som bidrar til lav kobling og høy samhørighet.



Figur 3-2: Backend Filstruktur

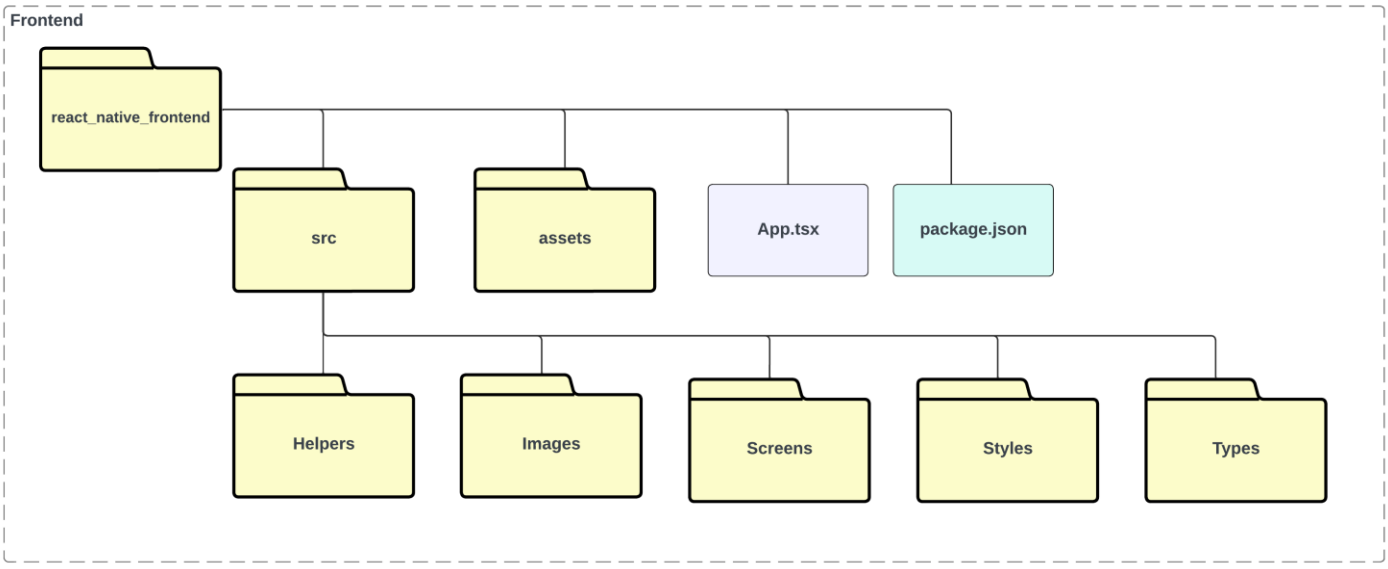
3.3 Frontend filstruktur

Figur 3-3: Frontend Filstruktur viser filstrukturen for klientsiden. react_native_frontend inneholder kode for frontend systemet og inneholder kode for brukergrensesnitt, HTTP forespørsler samt bruk av AI modellen. Src mappen inneholder houvedlogikken for applikasjonen og består av: Helper, images Screens, Styles og Types mappene. Helpers grenen brukes av andre klasser, og består av hjelpeklasser som håndterer transformasjon av data samt algoritmer som benyttes under utførelsen av beveglighetstestene. I tillegg til dette inneholder mappen logikk for oppkobling til HTTP, ved bruk av Ngrok og en simulert innloggingssystem- images inneholder bildene i applikasjonen. Screens mappen består av grensesnittet brukeren interagerer med, logikk for navigering mellom sidene samt kommunikasjon mellom frontend og backend.

Assets mappen inneholder de ulike AI modellene som systemet kan bruke, samt en liste over kroppsdeler og dets korresponderende punkt-id.

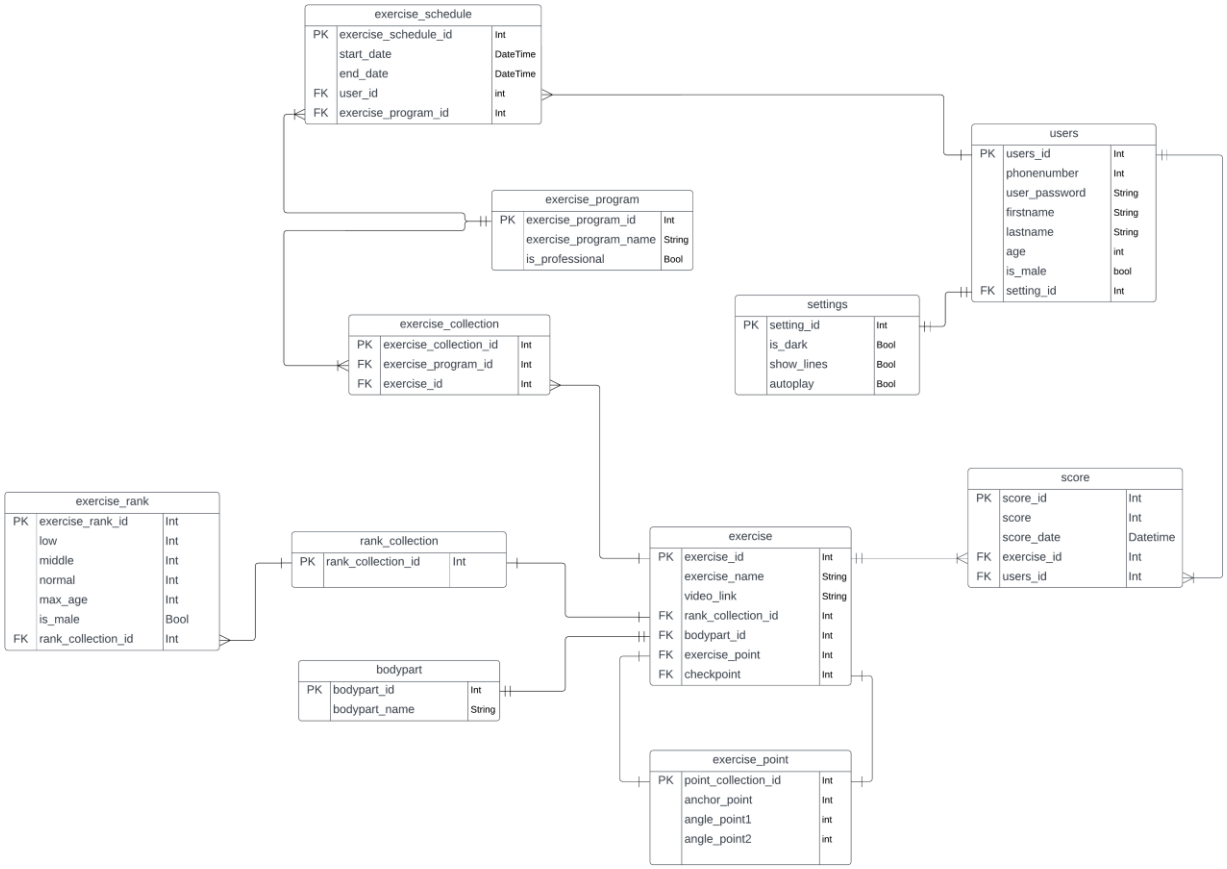
Styles mappen inneholder definisjonen av proporsjonene og designet til brukergrensesnittet. Types grenen mappen inneholder logikk for instansiering av objekter som benyttes av andre klasser.

App.tsx inneholder oppsettet for navigering i applikasjonen.



Figur 3-3: Frontend Filstruktur

4 DATABASEMODELL



Figur 4-1: ERD Database

Relasjonsdatabasen, som vises i Figur 4-1 benytter flere tabeller for å minimere mengden av duplisert lagret data. tabellen users lagrer informasjon om brukeren, imens settings lagrer brukerens preferanser for applikasjonen. Tabellene Exercise_program, exercise_schedule og exercise_collection lagrer data om oppfølginger, inkludert informasjon om når oppfølgingene skal starte og slutte, samt hvilke øvelser brukeren skal utføre under en oppfølgning. *exercise*, *bodypart* og *exercise_point* lagrer informasjonen om øvelsene, hvilken kroppsdel korresponderer til hver øvelse, samt hvilke punkter fra AI modellen benyttes for å beregne vinkelutslag. Etter endt test blir en score beregnet ut fra vinkelutslaget, resultatet baseres så på exercise_rank samt brukerens alder eller kjønn.

5 Server-tjenester

5.1 Endepunkt

Funksjon	Endepunkt	Forklaring	Type
GetBodyparts(int)	Bodypart/{id}	Henter en liste av alle kroppsdeler som brukeren har scores for.	Get
DeleteProgram(int)	EditList/{id}	Sletter en spesifikk liste av øvelser.	Delete
PutSelfDefined(int)	EditSelfDefined/{id}	Lagrer en ny selvdefinert test i databasen.	Put
GetSelfDefined(int)	EditSelfDefined/{exercise_program_id}	Henter ExerciseProgramName, id liste over aktive øvelser samt start og sluttdato for program	Get
GetExerciseProgramsForId(int)	ExerciseProgram/{id}	Henter aktive ExercisePrograms for en bruker	Get
GetExerciseRankForBodypart(int)	ExerciseRank/bodypart/{bodypartId}	Henter en ExerciseRank for en kroppsdel	Get
GetExerciseRankForExercise(string)	ExerciseRank/exercises/{exercises}	Henter ExerciseRanks for en liste av øvelser	Get
GetScoresForUser(int, int)	Score/{userId}/{bodypartId}	Henter alle Scores basert på bruker og kroppsdel	Get
PostScores(List<Score>)	Score/	Lagrer nye Scores i databasen	Post
receivePostSelfDefined(SelfDefinedReformat)	SelfDefined/	Lager ny oppfølging via nye ExerciseProgram, ExerciseCollection og ExerciseSchedule	Post
GetUserSettings(int)	Settings/{id}	Henter instillingene til en bruker	Get
PutUserSettings(Settings, int)	Settings/{id}	Oppdaterer instillingene til en bruker	Put

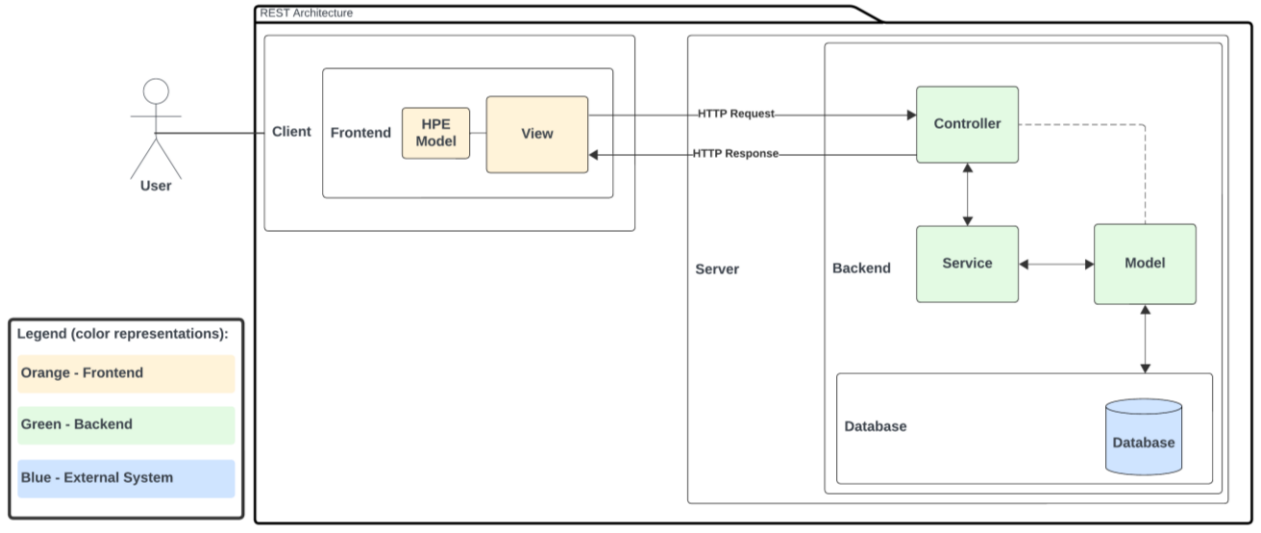
Grønn - Get
Rød - Delete
Gul - Put
Blå - Post

Figur 5-1: Endepunkter

Figur 5-1 viser filstrukturen for klientsiden og viser HTTP-endepunktene som klienten kan nytte for å kommunisere med serveren. Alle endepunktene starter med en URL eller IP adresse men dette er tatt vekk fra tabellen for å unngå redundans og fordi den ikke er statisk.

5.2 Rest Arkitektur

Systemet benytter en REST arkitektur basert på Server Client og HTTP forespørsler. Dette illustreres i Figur 5-2. Viewet i diagrammet representeres av Screens i programkoden. Brukeren interagerer med frontend grensesnittet, som i enkelte tilfeller sender HTTP requests til backend kontrolleren. Kontrolleren sender så forespørsler til Service laget som kommuniserer med databasen gjennom en dbcontext.



Figur 5-2: Rest Arkitektur

6 INSTALLASJON OG KJØRING

For å kjøre applikasjonen er det viktig å ha installert .Net versjon 8.0 og NPM, som kan lastes ned fra internett. dotnet_backend og react_native_frontend er utviklet ved hjelp av utviklingsverktøyet Visual Studio Code (VSC). Hvis VSC benyttes, må man ha noen ekstra pakker for å kjøre applikasjonen.

Her er en liste over utvidelsene som kreves for å kjøre applikasjonen og testene:

- .NET Core Add Reference
- .NET Core Test Explorer
- .NET Extension Pack
- C#
- C# Dev Kit

Etter å ha lagt til de riktige utvidelsene, kan terminalen i VSC benyttes for å navigere til *dotnet_backend* og deretter kjøre serveren med kommandoene *dotnet build* etterfulgt av *dotnet run*. Dette vil bygge koden og koble opp mot databasen.

Når "react_native_frontend" kjøres for første gang, må det navigeres til mappen med terminalen i VSC og kjøre kommandoen "npm install". Denne kommandoen vil automatisk laste ned alle pakkene som er nødvendig for å kjøre applikasjonen på mobilenheten.

Deretter kan applikasjonen bygges og kjøres ved denne kommandoen: "npm start".

Videre vil det komme en mulighet i terminalvinduet for å velge hvilken enhet man vil starte applikasjonen på. Figur 6-1, viser terminalvinduet.

Merk at vår nåværende fremgangsmåte kun tillater kjøring på Android-telefoner. For å kjøre på iOS så kreves det ytterligere trinn, inkludert tilgang til en MAC, som vi ikke hadde tilgang til under utviklingen av applikasjonen.

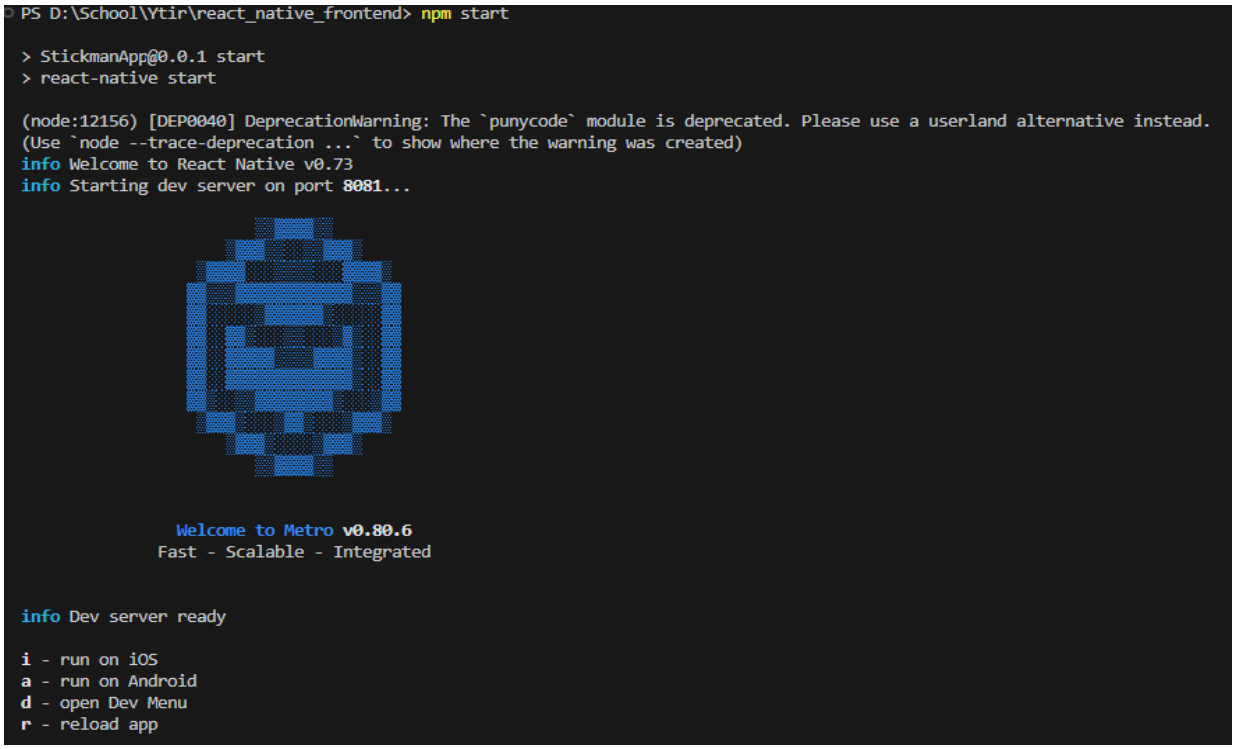
Når du skal kjøre applikasjonen på Android telefonen, må "USB-Debugging" være aktivert, ellers vil telefonen blokkere applikasjonen.

Det siste trinnet for å koble mobilapplikasjonen til backend er å sette opp riktig adresse for backend-APIet. Dette gjøres ved å inn i "react_native_frontend/src/GlobalVariables.ts" og endre "API_URL".

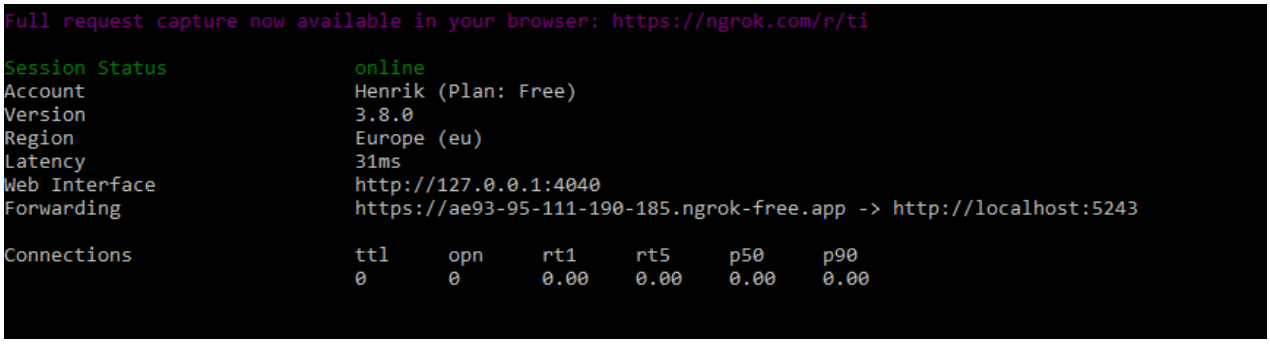
Siden backend delen av applikasjonen vår har blitt kjørt lokalt, har vi benyttet programvaren "Ngrok". Denne progarmvaren kan lytte til en port på datamaskinen og opprette en midlertidig gratis API-URL adresse som kan aksesseres via nettet.

For å opprette en slik URL, åpner man Ngrok og bruker kommandoen “ngrok http 5243”. Dette forteller Ngrok at den skal håndtere HTTP-trafikk på port 5243, som er porten backenden bruker når den kjører lokalt.

Så lenge både backenden og Ngrok kjører, kan man nå backend nås via HTTPS-koblingen som står i *Forwarding*-tabellen, se Figur 6-2.



Figur 6-1: Utførelse Av Program



Figur 6-2: Ngrok

7 DOKUMENTASJON AV KILDEKODE

Kildekoden som dokumenteres i dette kapitlet er ment å forklare hvordan kommunikasjonen mellom klient og server foregår. Bildene av koden viser eksempler på kjerne funksjonaliteter knyttet til systemet.

Figur 7-1 viser ett eksempel på en Get forespørsel på klientsiden (frontend). Get forespørsel forsøker å hente de relevante programmene til brukeren, får så å sette dataen til programs. Koden blir kjørt når brukeren laster inn siden.

Figur 7-2 viser ett eksempel på en Get forespørsel på server siden (backend) i applikasjonen, den mottar en bruker id, får så å gjøre ett kall til den korresponderende service klassen. Service klassen, som vises i Figur 7-3 henter så den relevante dataen fra databasen, for deretter å returnere den til kontrolleren. Kontrolleren sender så informasjonen til klientsiden (frontend).

```
const [programs, setPrograms] = useState([]);

useEffect(() => {
  fetch(`${API_URL}/ExerciseProgram/${USER_ID}`)
    .then(response => response.json())
    .then(data => {
      setPrograms(data);
    })
    .catch(error => console.error(`Error: ${error}`));
}, [isFocused]);
```

Figur 7-1: HTTP Get Frontend

```
namespace dotnet_backend.Controllers
{
    [Route("[controller]")]
    [ApiController]
    1 reference
    public class ExerciseProgramController : ControllerBase
    {
        1 reference
        private readonly AppContext _context;

        2 references
        private readonly ExerciseProgramService _service;

        0 references
        public ExerciseProgramController(AppContext context, ExerciseProgramService service)
        {
            _context = context;
            _service = service;
        }

        // GET: api/Exercise
        [HttpGet("{id}")]

        // NOTE -> HARDCODED user
        0 references
        public async Task<ActionResult<IEnumerable<ExerciseProgram>>> GetExerciseProgramsForId(int id)
        {
            var exercisePrograms = await _service.GetRelevantExercisePrograms(id);


            return exercisePrograms;
        }
    }
}
```

Figur 7-2: HTTP Get Controller Backend

```
namespace dotnet_backend.Services
{
    11 references
    public class ExerciseProgramService
    {
        5 references
        private readonly IDbContext _context;

        1 reference
        public ExerciseProgramService(IDbContext context)
        {
            _context = context;
        }

        2 references
        public async Task<List<ExerciseProgram>> GetRelevantExercisePrograms(int userId)
        {
            var currentDate = DateTime.Now;

            var exercisePrograms = await _context.Users
                .AsNoTracking()
                .Where(user => user.UsersId == userId)
                .Include(user => user.ExerciseSchedules)
                .ThenInclude(schedule => schedule.ExerciseProgram)
                .SelectMany(user => user.ExerciseSchedules)
                    .Where(schedule => schedule.StartDate <= currentDate && currentDate <= schedule.EndDate)
                    .Select(schedule => schedule.ExerciseProgram)
                .Distinct()
                .OrderByDescending(exerPro => exerPro.IsProfessional)
                .ThenBy(exerPro => exerPro.ExerciseProgramName)
                .ToListAsync();

            return exercisePrograms;

        }
```

Figur 7-3: HTTP Get Service Backend

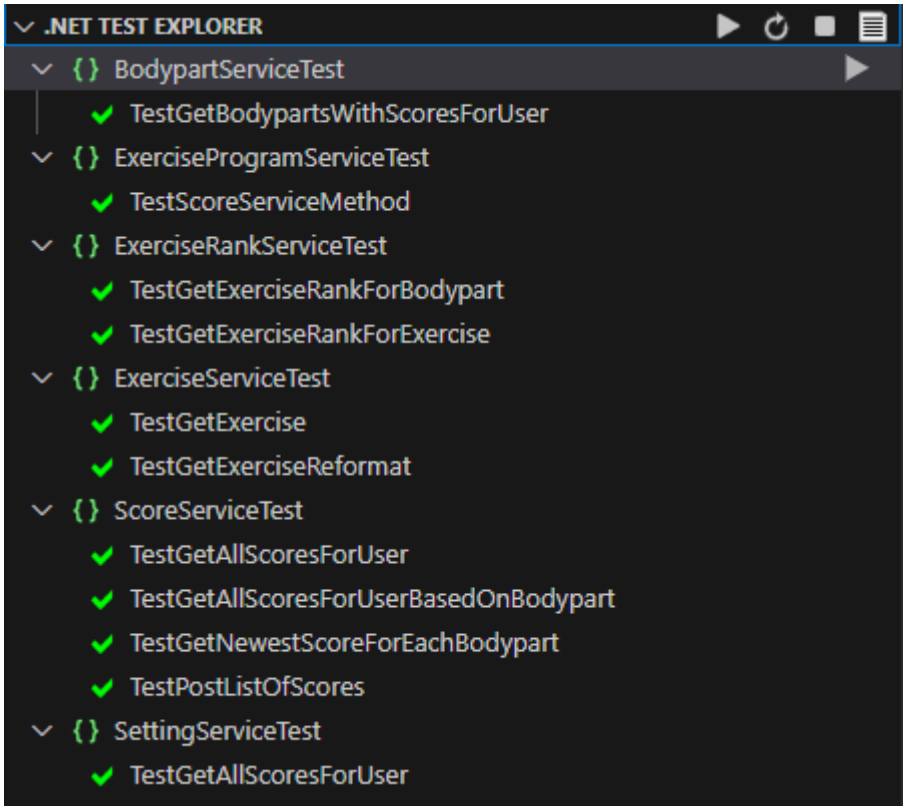
8 TESTING

I prosjektet benyttes det et dedikert testmiljø for å sikre en systematisk tilnærming til testing, noe som bidrar til en organisert og effektiv testprosess.

Testene utføres i VSC, og for å kunne kjøre testene i dette miljøet kreves det installasjon av de samme utvidelsespakkene som ble nevnt i kapittel 6. Etter å ha bygget med kommandoen "dotnet build" i terminalen, oppdages testene automatisk av *.Net Core Test Explorer*. Denne utvidelsen muliggjør kjøring av tester enten individuelt eller samlet, som vist i Figur 8-1.

Testing av serviceklassene er utført ved hjelp av en mock-database, dette reduserer risikoen for uønskede påvirkninger på den reelle databasen under testing av tjenestene. Denne tilnærmingen sikrer et isolert og kontrollert testmiljø, noe som bidrar til pålitelige og konsistente testresultat.

Testingen av frontend-komponentene har vært en manuell prosess, der automatiserte tester ikke er utviklet for systemet.



Figur 8-1: Enhetstester

9 REFERANSER