Projekt - GreetUp

Projektrapport

Emmy Lindgren
emli0277

Viktoria Nordkvist
vino0049

Christoffer Billman
chbi0025

Alex Wahlroos
alwa0043

30 maj 2022

Innehåll

1 Inledning

Det här projektet grundar sig i kursen *Teknik för social medier*. Bakgrunden är ett samarbete med marknadsföringsstudenter på Edith Cowan University (ECU) i Australien. Eftersom projektdelen i den kursen sammanföll samtidigt som projektet i denna kurs valde vi att slå ihop dessa till ett större projekt, vilket skulle möjliggöra ett mer välutvecklat system.

Problemformuleringen som gavs till den kursen var att social isolation ökat hos äldre till följd av pandemin. Målet var därför att utveckla ett socialt medium som ska verka för att bryta denna sociala isolation. Tillsammans med studenterna på ECU har vi tagit fram underlag för vad som bör ingå i en sådan applikation. Exempelvis bestämdes målgruppen, vilka behov dessa har och hur vi kan tillgodose dessa med applikationen.

Konceptet som togs fram landade i en evenemangsapplikation där användare kan skapa och delta i olika evenemang. Det kan till exempel handla om en promenad i skogen eller en picknick i parken. Då vår målgrupp består av äldre med funktionsvariationer är det viktigt att utforma applikationen därefter. Evenemangen som skapas är därför anpassade för att vara lämpliga för olika funktionsvariationer. Detta koncept är vad vi valt att implementera i vårt projekt som en mobilanpassad progressiv webbapplikation (PWA).

2 Problembeskrivning

Som nämnt i inledningen är detta en applikation anpassad för äldre med funktionsvariationer. Vår tidigare erfarenhet med denna målgrupp är bristfällig och av denna anledning är det viktigt att förstå och analysera användare, kunna se vilka behov och utmaningar de står inför och försöka tillgodose dessa genom applikationen.

Specifikationen för detta projekt var relativt öppen och stor frihet i val av implementation gavs. Temat för projektet var mashups. I praktiken innebär detta att utveckla en ny, eller bättre, webbapplikation baserat på diverse tjänster som andra utvecklat. Förslagsvis skulle detta göras med hjälp av publika API:er för att hämta data från olika hemsidor, och kombinera detta i ett eget gränssnitt. Efter diskussion med kursansvarig reviderades detta dock till att inte inkludera mashups, utan att vi skapar våra egna API:er. Detta eftersom vi redan utvecklat det koncept som ligger till grund för applikationen.

Inför projektet gjordes en teknikutvärdering av teknik som vi i projektgruppen ansåg relevant för just detta arbete. Detta var ett bra tillfälle att inhämta nödvändig kunskap för att kunna omsätta det till praktisk implementation i projektet. Vi valde alla fyra att undersöka olika områden och dessa var OAuth, React, AJAX och animationer på webben. Målet med teknikutvärderingen var alltså att implementera dessa tekniker i den färdiga applikationen.

3 Systembeskrivning

3.1 GreetUp

3.1.1 Pitch

Studenter från Umeå Universitet har tillsammans med studenter från Edith Cowan University tagit fram en applikation som ska hjälpa äldre med funktionsvariationer att känna sig mindre ensamma. Applikationen heter **GreetUp** och syftar till att de äldre ska ta sig ut och träffa nya människor i sitt område. **GreetUp** är ett enkelt hjälpmedel för att skapa aktiviteter och evenemang för människor i närområdet! De äldre får förslag på evenemang som passar just dem och deras funktionsvariationer.

Vi vill hjälpa och visa äldre att tillsammans är vi starka, och det är enklare än vad du tror att hitta nya vänner. Just **GreetUp**!

3.1.2 Funktionalitet

GreetUp låter användare skapa ett konto och logga in med hjälp av Google. Här förifylls information från Google så som bild och namn, men användaren kan enkelt ändra detta. Sedan ska även användaren ange sina funktionsvariationer (vilket såklart är privat) och sin adress.

Väl inne i applikationen kan användaren se evenemang som andra användare har skapat. Både privata och publika evenemang, men de privata visar begränsad information om plats om man inte är inbjuden. Man kan se information om evenemangen, anmäla intresse, klicka i att man ska gå och på de privata evenemangen kan man begära en inbjudan. På publika evenemang (eller privata evenemang som man är inbjuden till) kan man också se vilka andra användare som ska komma eller är intresserade av att gå.

Evenemangen visar också med hjälp av en symbol om de passar just användarens funktionsvariationer. En röd symbol varnar för att inga av funktionsvariationerna som användaren uppgett uppfylls av evenemanget, medan en gul flaggar för att några uppfylls. Ingen symbol visar på att evenemanget passar alla användarens funktionsvariationer. Användaren får alltså information om vad som kanske kan anses passa dem bäst, men valet är helt upp till dem om de vill gå eller inte.

Som användare kan man också skapa evenemang, där får man ange information, välja bilder och välja vilka funktionsvariationer evenemanget passar för. Kategorier för evenemanget väljs också, och ifall det ska vara publikt eller privat.

Väljer användaren att skapa ett privat event måste denne acceptera andra användare som har begärt inbjudan för att de ska få all information om eventet och läggas till i “närvarar”-listan.

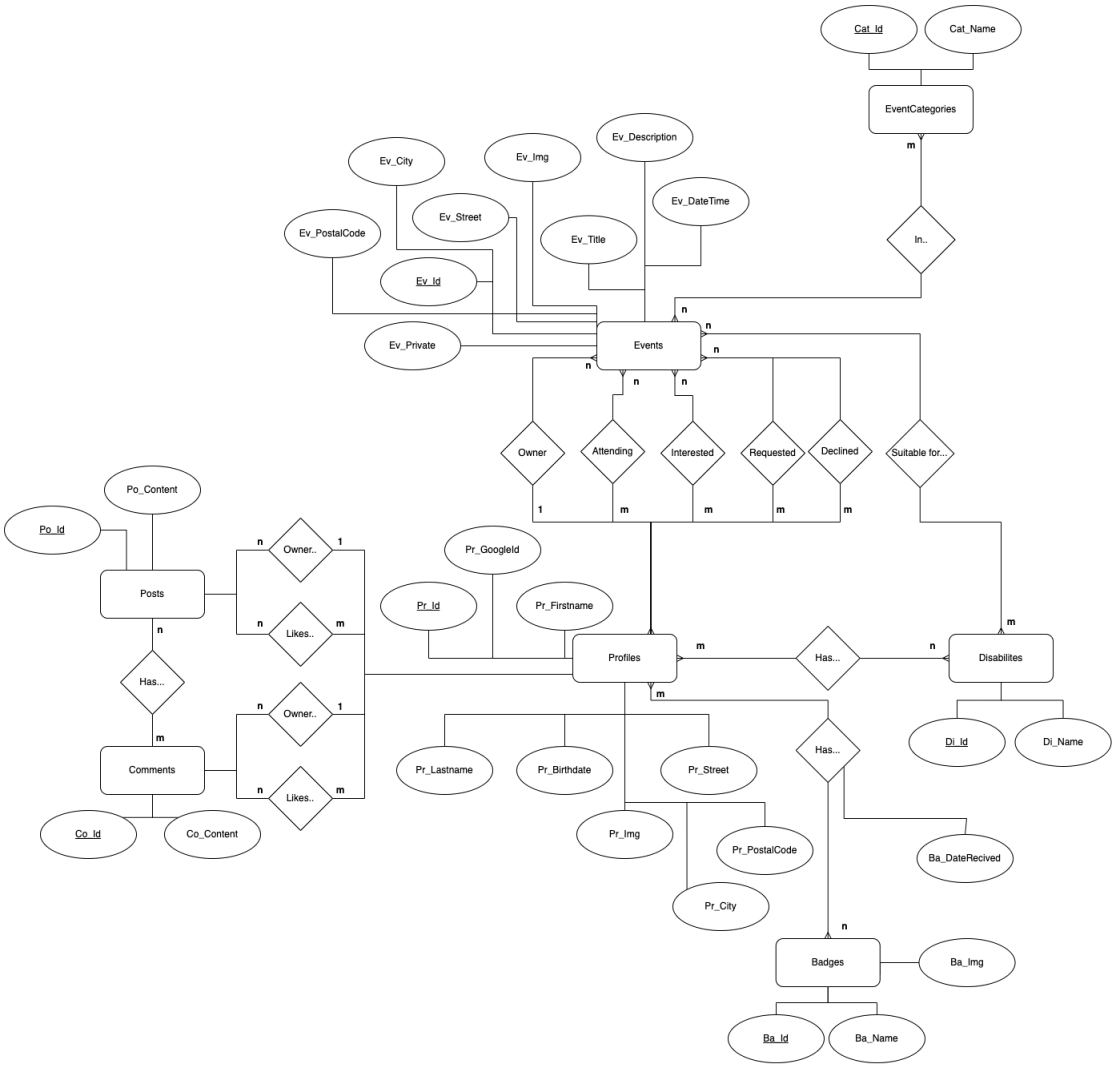
Användaren har också ett “eget schema” som bara visar de evenemang som användaren äger eller har anmält att de ska gå på. På så sätt blir det enkelt att hålla koll på vilka evenemang som man faktiskt ska delta i.

Det finns även en *Neighbourhood wall* där användare kan göra, kommentera och gilla inlägg. Det ska fungera som en central och säker plats för de äldre att kommunicera på med de närmsta i deras omgivning. Det kan också fungera för att de ska lära känna varandra och samtala innan evenemang faktiskt äger rum.

Till sist finns även en *Gamification*-funktionalitet som innebär samling av så kallade *Badges*. Tanken är att användare kan samla poäng genom att göra olika saker i applikationen. Exempelvis gå på många evenemang inom samma kategori, skapa nya event, göra många inlägg på *Neighbourhood wall* och så vidare. Med hjälp av dessa poäng kan man då samla på sig olika *Badges* beroende på vilka poäng det är. Så, genom att vara en aktiv användare får man *Badges*!

3.2 ER-Schema

Databasen som ligger till grund för detta projekt innehåller många tabeller med olika relationer och attribut, där de mest omfattande tabellerna är de för användare och evenemang. Dessa har i sin tur många relaterade tabeller som utgör stommen i databasen. En representation av ER-schemat presenteras i Figur 1.



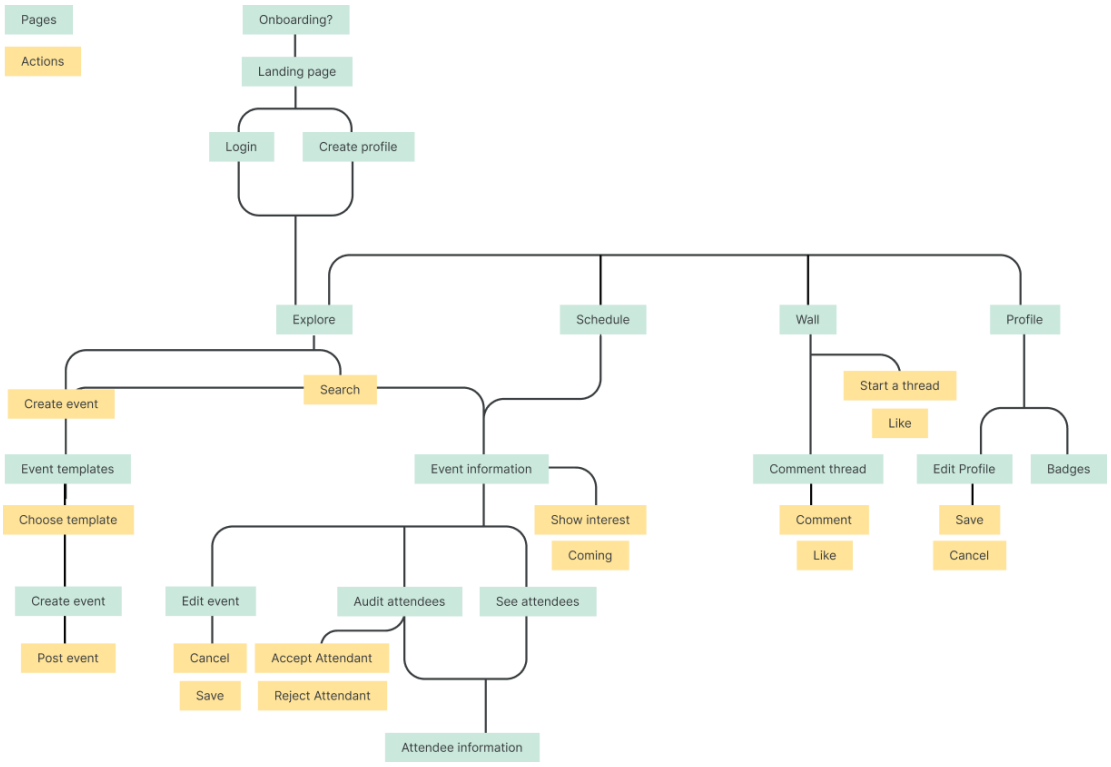
Figur 1: ER-schema av databasen

Rutorna i ER-schemat representerar tabeller och cirklarna representerar attribut i dessa tabeller (kolumner). Exempelvis har tabellen *Profile* atttibut som Id, FirstName, LastName och så vidare. Romberna i schemat representerar relationer mellan olika tabeller och är således ofta ett attribut i en tabell som refererar till en annan tabell. Exempelvis

innehåller tabellen *Event* relation till *Categories* för representation av vilka kategorier som evenemanget tillhör.

3.3 Flödesschema

För att åskådliggöra navigationen i gränssnittet skapades ett flödesdiagram, eller flödesschema. De gröna rektanglarna representerar de sidor som visas för användaren och de gula rektanglarna representerar vad användaren kan göra, det vill säga, vilka åtgärder som kan genomföras. Detta representeras i Figur 2.



Figur 2: Flödesdiagram för gränssnittet

3.4 Säkerhetsåtgärder

En utmaning som vi tidigt ställdes inför var applikationens säkerhet givet målgruppen. Vi vet att många äldre utsätts för bedrägerier online där exempelvis uppgifter om bankkonton, lösenord och annan känslig information riskerar att hamna i fel händer.

Detta kom att prägla vår utformning av applikationen, där exempelvis chatt- och samtalsfunktioner inte finns eftersom detta skulle kunna leda till oönskad kontakt. Huvudsyftet är inte heller att kommunicera på applikationen utan träffas i det verkliga livet och därför är dessa funktioner inte eftersträvansvärda att inkludera. En annan säkerhetsåtgärd som vidtogs var att, i de fall det är ett privat evenemang som hålls, är platsen där detta sker

inte synligt för de som inte blivit accepterade till evenemanget. På så sätt kan den som skapat evenemanget känna sig trygg i att platsinformationen som delas om evenemanget inte når ut till obehöriga, om det exempelvis skulle vara hemma hos någon.

För att användare ska kunna känna sig ännu tryggare fyller även *Badges*- funktionaliteten ett syfte. På den lista där man kan se vilka som ska gå på ett evenemang kan användaren välja att se mer information om personerna som ska gå. Då får man se namn, vilken stadsdel de är ifrån och vilka *Bagdes* den personen har. *Bagdes* kan man som sagt bara få ifall man är en aktiv användare, vilket betyder att om en person har många *Bagdes* så kan man lita på att personen är aktiv, och faktiskt gör saker på applikationen.

Tekniska säkerhetsåtgärder som vidtagits är att användare registreras och loggar in via sitt Googlekonto. Detta innebär ett extra steg för någon som vill utge sig för att vara någon den inte är. Vi förvarar inte heller några lösenord i databasen, utan använder istället ett ID kopplat till varje Googlekonto. Detta ID hashas och saltas innan det sparas i databasen för extra säkerhet. En sista åtgärd som vidtagits är att säkra upp API:et med API-nyckel som varje registrerat konto får tillgång till. På detta sätt kan inga obehöriga komma åt data ur databasen, utan bara registrerade användare som loggat in.

4 Implementation

Databasen, implementerad med Entity Framework (EF), hanteras i mySQL. För att kunna interagera och kommunicera med databasen har egna API:er skapats och dessa används uteslutande för detta ändamål. Genom implementation av AJAX, mer specifikt Axios, kan vi smidigt sköta kommunikationen med databasen. Vår back-end är byggd i .NET medan front-end är skapad med JavaScript-biblioteket React.

4.1 Back-end

Databasmotorn som används är MySQL Server. Detta i och med att majoriteten av teamet arbetar från macOS och det var därför ett krav att det skulle fungera på macOS såväl som Windows. Av denna anledning räcker inte Microsoft SQL Server, som enbart fungerar på Windows.

4.1.1 Databas

För att skapa databasen utgick vi från *Code-First-principen* med hjälp av Entity Framework. *Code-First* innebär att man först skapar modeller i koden som innehåller de attribut som man vill att tabellerna ska ha. En tabell i databasen (alltså en ruta i ER-Schemat) motsvarar en modell. Exempelvis kan vi titta på en bit av koden för modellen `ProfileModel` som sedan skapade tabellen *Tbl_ Profiles* i databasen.

```
[Table("Tbl_Profiles")]
public class ProfileModel
{
    [Key]
    public int Pr_Id { get; set; }
    public string Pr_Firstname { get; set; } = String.Empty;
    public string Pr_Lastname { get; set; } = String.Empty;
    .
    //One-to-many relationship
    public virtual ICollection<ProfileDisabilityModel>?
                        Pr_Disabilities { get; set; }
    .
    //Many-to-many relationship
    public virtual ICollection<AttendingModel>?
                        Pr_AttendingModel { get; set; }
    .
    .
}
```

Figur 3: Kod för modellen `ProfileModel`

Här kan man se att attribut som ska finnas med i databasen listas som attribut i modellen.

För att skapa *en-till-många relation* mellan två tabeller lägger man även detta som ett attribut i modellen som är på "en-sidan" av relationen, den som har "många" av den andra tabellen. Den tabell som refererar till en annan kallas *barn* medan den som refereras till kallas *förälder*. I Figur 3 ovan visas att `ProfileModel` har en *en-till-många relation* med tabellen `Disabilities` för att representera de funktionsvariationer som personen har.

För att skapa *många-till-många relationer* mellan tabeller krävs det dock att man gör en enskild modell för detta, i databasen blir detta också en egen tabell. Så som relationen mellan evenemang och vilka som ska komma på evenemangen, *Attendees*. Den modellen innehåller då ett attribut för var och en av modellerna som ska länkas, samt ett attribut för deras huvudnycklar. Nedan ser vi kod för modellen *Tbl_Attendees* som länkar ihop evenemang och profiler som ska gå på dessa.

```
[Table("Tbl_Attendees")]
[Keyless]
public class AttendingModel
{
    public EventModel At_Event { get; set; }
    public ProfileModel At_Profile { get; set; }

    public int Ev_Id { get; set; }
    public int Pr_Id { get; set; }
}
```

Figur 4: Kod för modellen `ProfileModel`

Sedan måste även modellerna som berörs (i detta fall `Profiles` och `Events`) innehålla ett attribut som länkar till denna modell. Det kan ses i koden för `Profiles` som hittas i Figur 3.

När alla modeller skapats måste även en `DbContext`-fil skapas. Denna definierar vilka modeller som faktiskt ska med till databasen, det vill säga vilka som ska bli tabeller. I filen `appsettings.json` måste man även ange en *connection string*, i den anger man vad man vill att databasen ska heta och var den ska skapas.

Sedan kan databasen genereras. Då används `Package Manager Console` för att skapa så kallade *Migrations*, vilket är en fil som genererar databaskod utifrån de modeller som skapats. Efter detta körs kommandot `Update-Database` och databasen skapas.

Ifall man efter detta vill göra någon förändring i databasen (modellerna) måste man göra en ny migration och uppdatera databasen igen.

4.1.2 API

Med fördel kan även API-kontrollers med CRUD autogenereras för de flesta tabeller/modeller när man använder sig av Entity Framework. I detta projekt har dessa kontrollers genererats, men ändrats ganska mycket i linje med våra behov.

I Visual Studio finns hjälpmedel för att generera API-controllers, men eftersom vi utvecklade i Visual Studio Code fanns inte den möjligheten. Istället kan kod autogenereras med hjälp av terminalen (Microsoft, 2022). Först måste `code-generator` från Microsoft installeras sedan kan flera kod-alternativ genereras. För att generera en API-controller används följande kommando:

```
dotnet aspnet-codegenerator controller -name BadgeController -m BadgeModel
-dc Database -relativeFolderPath Controllers -useDefaultLayout
-referenceScriptLibraries -useAsyncActions -restWithNoViews
```

Då autogenereras en controller utifrån databasen och modellen som är vald.

4.1.3 Säkerhet

Angående säkerhet har en egen säkerhetslösning implementerats. En klass *AuthHandler* har skapats vars huvudsakliga uppgift är att jämföra innehållet i en HttpHeader *ApiKey* med en post *ApiKey* i databasen hos användaren. Metoden som gör detta heter *Authenticate*, och kallas i början av alla API-actions där autentisering ska krävas. Överensstämmer dessa vet vi att användaren är den som den utger sig vara. Denna HttpHeader ges till användaren vid registering av konto eller lyckad inloggning. Den skickas då med i varje request mot API:et.

Mer ingående, så består ApiKey av användarens id i databasen, ett understreck (_), sedan användarens hashade och saltade GoogleId.

En nackdel som finns är att användares GoogleId också sparas ohashat i databasen. Detta för att den ohashade versionen krävdes i en del av inloggningsprocessen. Detta är inte optimalt, och enligt Google själva så bör inte det oshashade GoogleId sparas under några omständigheter (Google Identity, 2022). Google har dock på senare år begränsat vad användare kan göra med ett GoogleId och med hjälp av GoogleId kan endast publik data hämtas. Själva Id:t går säkerhetsmässigt att likna med en mailadress eftersom det är detta som Id:t representerar. Denna aspekt är dock något som skulle behöva undersökas närmare vid vidareutveckling.

4.2 Front-end

4.2.1 React

Front-end programmeringen skedde med hjälp av ett JavaScript-bibliotek som heter React. Det är gjort primärt för utveckling av gränssnitt och koden som skrivs körs direkt i webbläsaren. Det som byggs är en single-page-app, SPA, med endast en HTML sida och sedan sker routing med React. React är uppbyggt av komponenter och de fungerar likt funktioner i andra språk, men i React returnerar de HTML-element. React använder sig av en virtuell DOM som är en stor anledning till att applikationer byggda med React är snabba på att dynamiskt rendera ifall något ändras på sidan.

För detta projekt användes version 17.0.2 av React samt version 6.3 av Reacts inbyggda routingsystem React Routing.

4.2.2 Google-login

Inloggningen på applikationen sker med hjälp av Google för att göra applikationen säkrare. Detta innebär att man måste ha ett Google-konto innan man kan skapa en användare på vår applikation vilket ökar säkerheten för våra användare. För att kunna implementera detta behövde projektet utökas med ett npm-paket som heter `react-google-login` som har färdigbyggda knappar och funktioner för att logga in med Google. Det som behövdes göras för att detta skulle fungera var att ett Client ID behövdes hämtas för att genomföra denna lösning. Denna inloggning skedde även med OAuth och för att göra detta följdes en instruktion (Freaky Jolly, 2022) som stegvis förklarade hur detta skulle implementeras.

I stora drag behövdes det skapas ett projekt på Google Cloud Platform, sedan skapades en OAuth Consent Screen för att få tillgång till ett OAuth Client ID som sedan behövs för att Google-login knapparna från paketet vi använde oss av skulle fungera. Sedan installerades `react-google-login` paketet och via det kunde komponenterna *GoogleLogin* samt *GoogleLogOut*, som är färdig implementerade knappar som innehåller logiken bakom uppkopplingen till Google samt hämtningen av data, användas.

4.2.3 Komponenter

Komponenter kan som tidigare nämnt liknas med funktioner i andra språk och de gör att kod enkelt kan återanvändas då dessa komponenter kan nås från andra komponenter och det kan skapas flera instanser av dem. Det finns två olika typer av komponenter, klasskomponenter och funktionskomponenter. I tidigare versioner av React var det vanligare att använda sig av klasskomponenter då de kunde ha tillstånd (eng. state). Men i senare versioner kan även funktionskomponenter ha tillstånd och använda sig av s.k. hooks vilket har lett till att det är vanligare idag att använda sig av funktionskomponenter. I början av detta projekt användes och skapades det båda typer av komponenter men detta ändrades till att alla komponenter var funktionskomponenter för att hålla det konsekvent.

Komponenterna som skapats under projektets gång har delats in i två övergripande kategorier, sidor som ska finnas i applikationen samt komponenter som finns på de sidorna. Sidor är de olika webbsidor som finns i applikationen, dessa komponenter kallar och använder sig mest av andra komponenter som i sin tur bygger upp sidan.

För att skicka data mellan komponenter används tre olika metoder. Den första metoden är att använda det inbyggda nyckelordet *props* som fungerar liknande som parametrar och argument i programmeringsspråk. En extra del som byggts på React är react-router-dom (React Router, 2022), som gör det möjligt att skapa sidor som beter sig som i den traditionella webben. När man då byter mellan olika sidor kan inte props användas för att skicka data mellan dem. Då används den andra metoden att skicka data, *location states* för att kunna skicka parametrar till komponenten som ska omdirigeras till. För att omdirigera och skicka denna data används hooken *useNavigate*.

```
const navigate = useNavigate()
navigate('sida', {state: data})
```

För att ta emot en location state-parameter måste komponenten som dirigeras till, använda sig av en hook som heter *useLocation* som hämtar location state. Den tredje metoden som används för att skicka information mellan komponenter är en hook som heter *useImperativeHandle*. Props och location-state kan enbart skicka data nedåt till barn i komponentträdet. Med hjälp av useImperativeHandle (och *forwardRef*) så kan man definiera funktioner i komponenter, som är nåbara utifrån komponenten. Exempelvis kan en förälderkomponent hämta information som enbart finns i en barnkomponent med hjälp av en sådan. Däremot är denna metod bara lämplig när man har komplex logik och många tillståndsvariabler som inte praktiskt kan "lyftas upp" (React, 2022a) till föräldrakomponenten, och skickas med som props istället. Generellt är det lite av ett "anti-pattern" i React att skicka data uppåt i komponentträdet, och bör i de flesta fall undvikas. Det finns även ett fjärde sätt att dela data mellan komponenter, som inte använts i detta här projektet, och det är s.k. contexts. Contexts kan tänkas som variable-scope i vanlig programmering. Man omger ett block av komponenter med en s.k. context, och då görs alla variabler som finns inom det blocket, tillgängliga för alla komponenter, utan att de behöver skickas med som props. Man skapar en context med funktionen React.CreateContext, och inkluderar en context i en komponent med hjälp av useContext-hooken (React, 2022b).

Många av de komponenter som skapats gör anrop till API:et för att hämta data direkt ur databasen för att använda i komponenten och visa för användaren. Mindre komponenter som endast behöver en specifik lista eller några få parametrar gör inte egna API hämtningar utan de tar in den information de behöver från förälderkomponenten.

Exempel på komponenter i applikationen

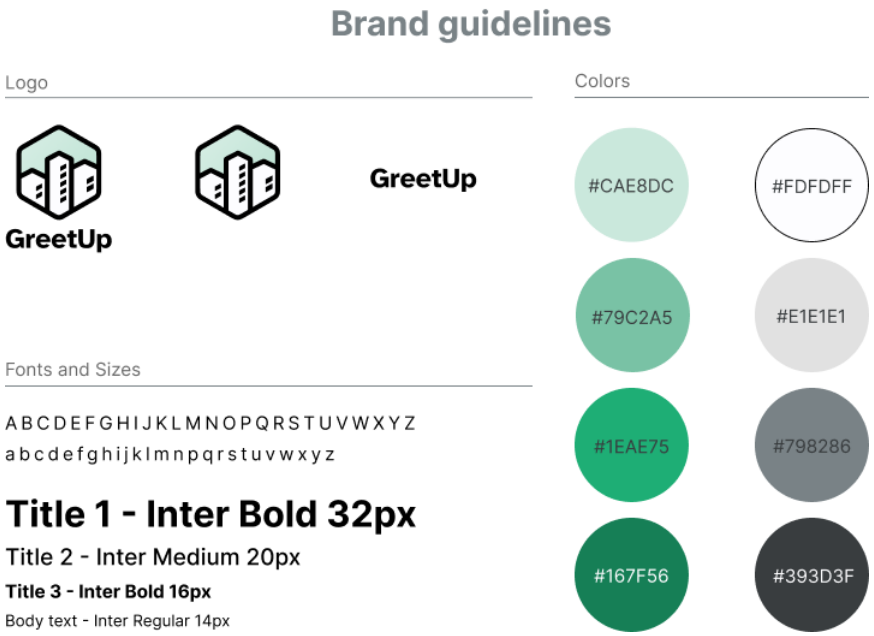
Knappar som finns i applikationen är ett bra exempel på hur vi valt att använda oss av komponenter. Knapparna som designats för HiFi prototypen i Figma var relativt lika varandra, därför togs beslutet att det skulle göras en anpassningsbar komponent kallad *Button*. Den skulle ta in vilken färg knappen skulle ha, vad som skulle stå som text på den, vad som händer om man klickar på den samt vilken ikon som skulle finnas med om det skulle vara en ikon på knappen. Dessa parametrar skickas från den komponent som vill använda sig av en knapp och de skickas med hjälp av *props*. Knappkomponenten tar då till exempel in parametrarna; *grön, Godkänn, bock-ikon, godkänn person*. Detta betyder då att det ska skapas en grön knapp där det står "Godkänn" och på den knappen ska det finnas en ikon i form av en bock och när man klickar på knappen kommer funktionen godkänn person köras. Beroende på hur knappen ska se ut och fungera så skickas olika parametrar in till komponenten som då själv innehåller logik för att skapa en knapp ut efter de parametrarna.

En annan typ av komponent som har skapats är en som representerar en hel sida i applikationen. Ett exempel på en sådan komponent är *Explore*, den hämtar all data den behöver som ska visas för användaren genom hämtningar från API:et. Dessa hämtningar görs med hjälp av ett bibliotek som heter Axios som skapar HTTP-förfrågningar. I komponenten Explore görs en förfrågan via API:et till databasen där komponenten ber om samtliga kommande evenemang, som då sparas i en lista som används för att skapa likadana kort på sidan för varje evenemang. Det som sker då är att i komponenten Explore finns en funktion som går igenom listan med samtliga evenemang och för varje evenemang i den listan skapas en komponent, *EventCard*, som visar informationen om det specifika evenemanget. På detta sätt skapas en mängd EventCard-komponenter på Explore sidan som visar alla evenemang för användaren. Komponenten EventCard tar in ett evenemang och ett tillstånd som parametrar, informationen ur evenemanget plockas ut och visas på samma sätt för alla evenemang. Tillståndet används för att se till att API-förfrågan har skett innan vi skickar vidare ett evenemang som skulle kunna vara tom. Om tillståndet följer med som "laddat" så kommer kortet som visar evenemanginformationen att renderas.

5 Gränssnitt

5.1 Grafisk profil

Till detta projekt skapades en grafisk profil för att kunna åstadkomma en enhetlig stil på applikationen. Typsnitt och dess storlekar valdes med målgruppen i åtanke, då det krävs att de är tydliga och läsbara. Färgerna är av god kontrast och klickbara ytor är enhetliga i största möjliga utsträckning.

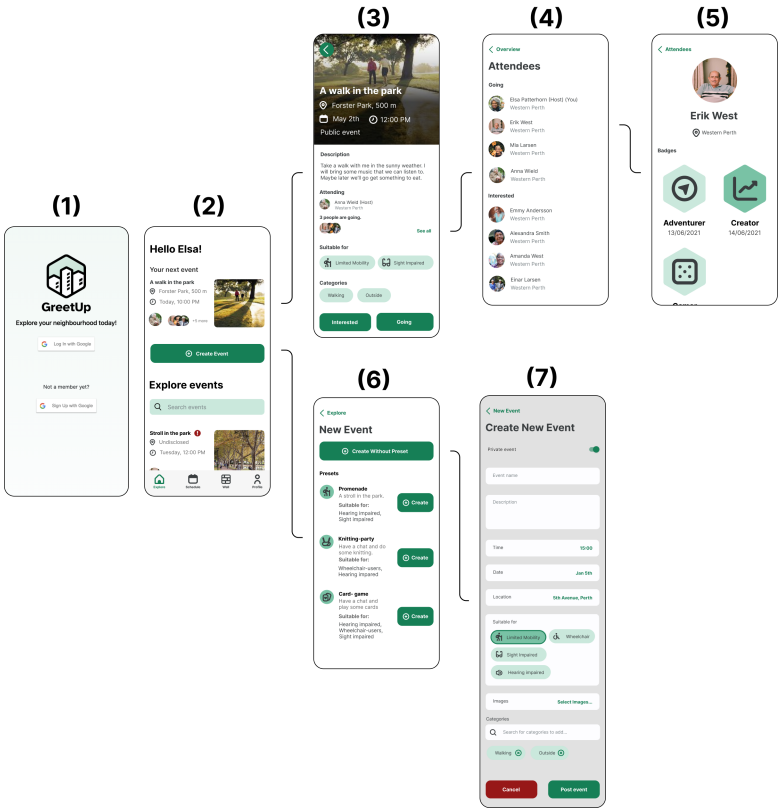


Figur 5: Grafisk profil

5.2 Förklaring av gränssnitt

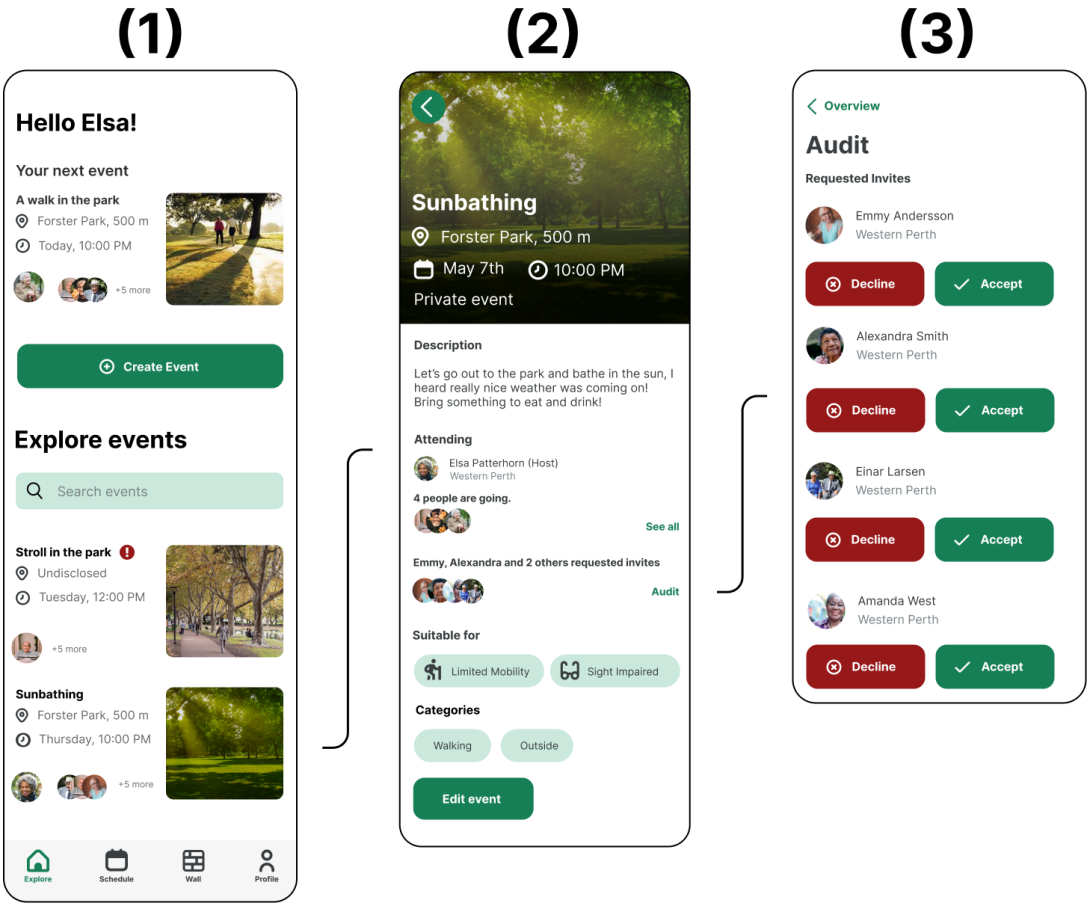
På loginskärmen (1) kan en användare välja att skapa en profil genom sitt Googlekonto eller logga in om denne redan är en registrerad användare. Då kommer användaren till startsidan, eller Explore, (2). Här visas alla kommande evenemang och kort information om dessa, samt att användaren ges möjlighet att söka på kommande evenemang. Vill användaren se mer information om ett publikt evenemang klickar denne på ett sådant och kommer då till en sida med mer information (3). På denna sida finns en mer utförlig beskrivning och även en lista på de som kommer. Klickar användaren på den listan kommer denne till en ny sida (4). Härifrån kan användaren klicka på en specifik deltagare för att de kort information om denne, inklusive vilka märken denne har (5).

Vill istället användaren skapa ett nytt evenemang görs detta också från Explore. Då klickar användaren på den gröna knappen och kommer till skärm (6). Härifrån kan användaren välja att skapa evenemang utifrån en mall eller skapa utan en mall. Skapar användaren utan en mall klickar denne på den stora gröna knappen *Create event* och kommer till skärm (7). Härifrån kan då information om evenemanget fyllas i och därefter kan evenemanget publiceras. Se Figur 6.



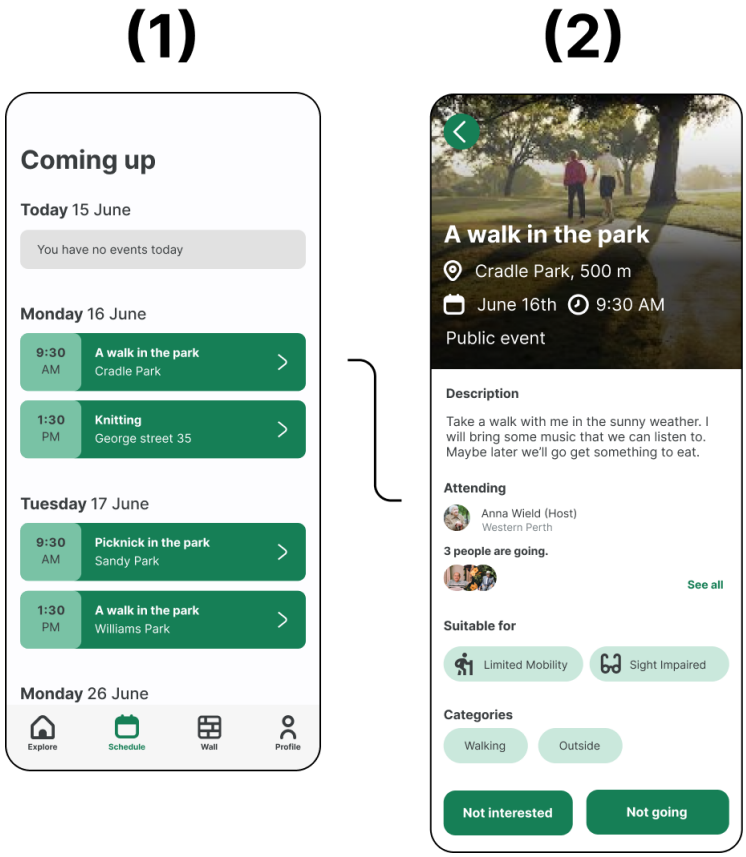
Figur 6: Explore för publika evenemang

Går användaren istället in på sitt eget privata evenemang från hemskärmen (1) kommer denne till informationssidan för det evenemanget (2). Härifrån kan användaren, som skapare av evenemanget, dessutom ändra information om evenemanget och se vilka som skickat förfrågan om att komma samt godkänna eller neka förfrågningar (3). Se Figur 7.



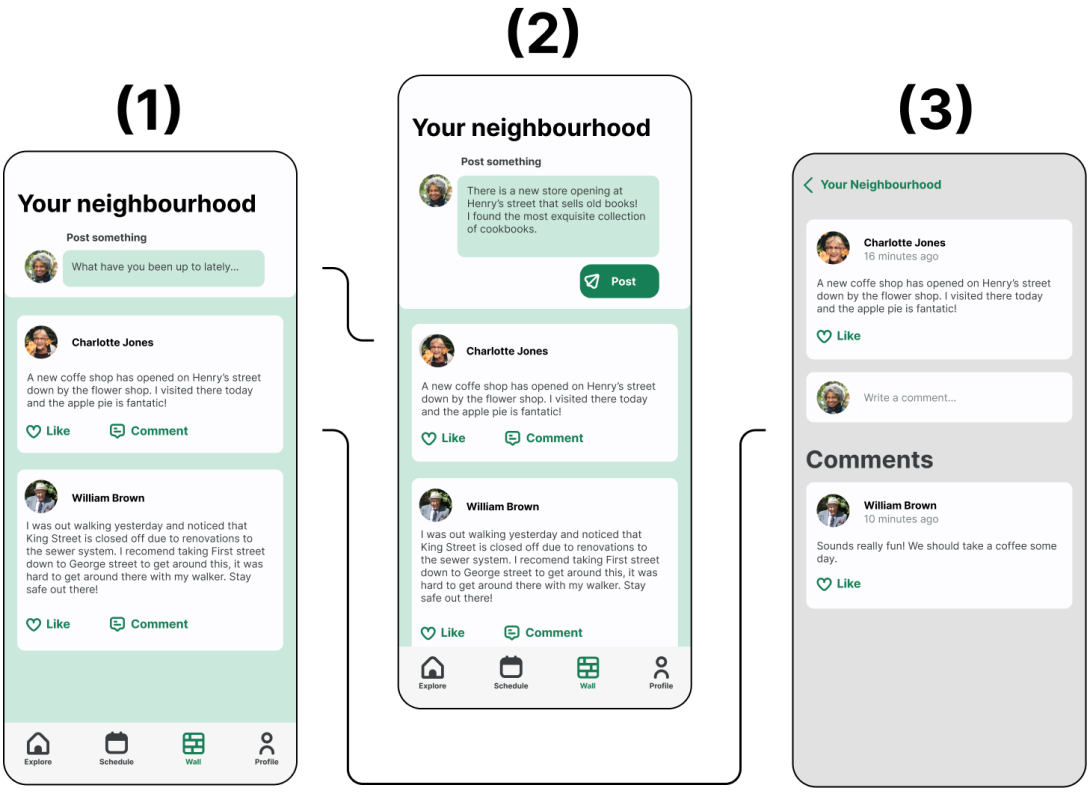
Figur 7: Explore för private evenemang som host

Om användaren navigerar vidare från “Explore” till “Schedule” kommer denne till (1). Där visas de evenemang som användaren anmält att denne kommer delta i samt om det är något evenemang som inträffar idag. Det går även att visa mer information om dessa evenemang genom att klicka på det och användaren kommer då till (2). Se Figur 8.



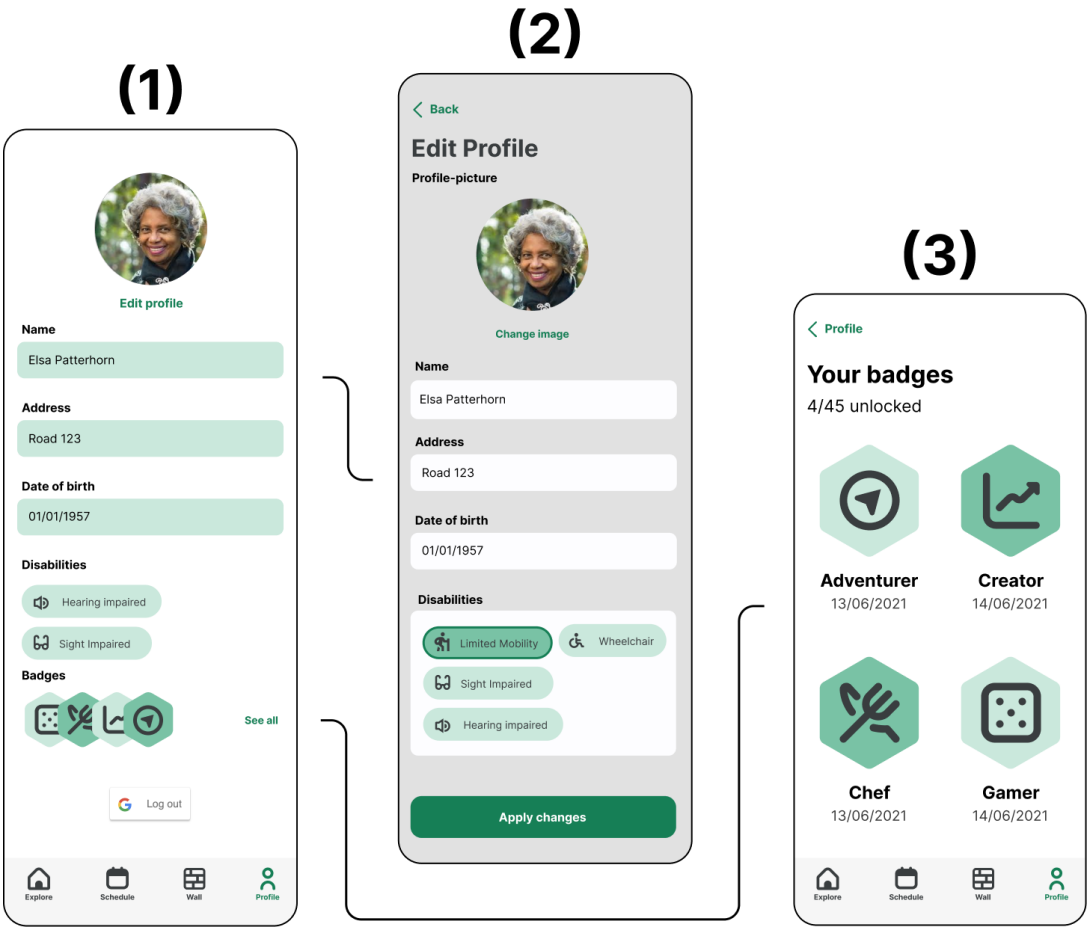
Figur 8: Schedule

Navigerar användaren till "Wall" kommer denne till (1). Här ges en översikt av alla inlägg som användare skapat. Användaren kan skapa inlägg genom att klicka på den ljusgröna knappen under *Post something* och kommer då till (2) där inlägget kan skrivas och postas. I Wall kan inlägg både gillas och kommenteras, och klickar användaren på *Comment* kommer denne till (3). Där kan användaren se andras kommentarer eller skicka en egen. Se Figur 9.



Figur 9: Wall

Navigerar användaren till "Profile" finns det personliga uppgiter såsom namn, adress, födelsedatum och vilka funktionsvariationer användaren vill att evenemang ska filtreras på. Härifrån kan även användaren välja att logga ut. Användarens personuppgifter går att ändra genom att klicka på *Edit profile* och användaren kommer då till (2). Klickar användaren istället på *See all* vid Badges kommer denne till (3) och kan se vilka märken som denne har fått samt information om när. Se Figur 10.



Figur 10: Profile

6 Lösningens begränsningar

Den första, och enligt oss största, förenklingen som kommer tas upp behandlar avstånd till ett evenemang. I vår urspringsidé var tanken att evenemang sorteras automatiskt på avstånd. Det vill säga, de evenemang som ligger närmast användarens nuvarande plats kommer att visas först. Så blev inte fallet och i brist på tid var detta en förenkling som var nödvändig eftersom det hade varit väldigt tidskrävande att utveckla sådan funktionalitet.

Något vi ville implementera i vår applikation var "gamification" (eller "spelifiering" på svenska). Detta innebär i praktiken något inslag av ett spelelement där principer för spel används i områden där spel inte traditionellt använts. Vår tanke var att implementera märken som användare kan samla och visa upp som samlingsobjekt för andra. Detta implementerades också till viss del, men vi utelämnade att implementera *hur* användare skulle få tillgång till dessa märken. Exempelvis skulle ett märke kunna vara att en användare gått på 10 evenemang, men i dagsläget har vi alltså inget som håller koll på statistik kopplat till detta.

En annan sak vi missade under utvecklingsfasen var att inkludera något sätt att visa de evenemang användaren själv anmält intresse för, det vill säga att användaren inte anmält att denne kommer, utan endast att denne är intresserad av evenemanget. En vidareutveckling skulle inkludera en sida som kan visa ens egna evenemang som man visat intresse för, eftersom en intresseanmälan endast visas på ett specifikt evenemangs informationssida.

Om en användare vill se information om evenemang som denne tidigare deltagit på är detta inte något som går att göra i dagsläget. En lösning på detta skulle kunna vara att implementera en till flik under sidan "Schedule" som innehåller gamla evenemang som redan varit.

Ännu ett förslag på vidareutveckling är hur evenemang presenteras på sidan "Explore". Grundidén var att sortera evenemang baserat på de funktionsvariationer användaren har. Det vill säga, evenemang som är lämpliga för användaren kommer först (som sedan också skulle sorteras baserat på avstånd till evenemang som tidigare nämnt), medan evenemang som inte är lämpliga givet en användares funktionsvariationer kommer sist. En förenkling, om än inte nödvändigtvis en försämring, som gjordes var att visa alla evenemang, men visa de evenemang som inte var anpassade givet funktionsvariation med en varningssymbol som indikerar att det kanske inte är lämpligt att gå på detta evenemang. I en vidareutveckling skulle en lösning kunna vara att användaren själv får välja hur denne vill visa och sortera evenemang.

Till sist anser vi också att inloggningsmöjligheterna bör uttökas till fler än Google. Detta för att utöka möjligheten att bli medlem och göra det så enkelt som möjligt för de äldre att använda sig av applikationen. Främst syftar vi då på möjligheterna att kunna logga in med exempelvis Facebook eller Apple, men också en säker "vanlig" inloggning bör finnas.

7 Problem och reflektioner

7.1 Problem

Till att börja med var React för oss ett ganska, om inte helt, främmande tillvägagångssätt för att bygga front-end. Många av de problem vi stött på har därför varit kopplade till just detta. Eftersom vi inte hade mycket erfarenhet var många av problemen kopplade till att vi använde funktioner och liknande på fel sätt, eftersom vi saknade denna grundförståelse. Med projektets gång kunde vi dock lära oss alltmer och de grundläggande, enkla problemen blev allt färre, vilket förstås är ett bra kvitto på att vi lärt oss mer än vi kunde tidigare. Alla problem var dock inte triviala och vissa fastnade vi lite längre med.

Till exempel hade vi problem med den automatiska uppdateringen av sidor på applikationen. Exempelvis om man tycker på knappen *Going* under ett event så skulle denna uppdatera databasen (vilket vi fick till) men också uppdatera knappens text till *Not going* och även uppdatera *Attending*-listan i vyn.

I början stötte vi på problem med SSL-certifikatet för hemsidan. I vanliga fall så skapar node/React automatiskt ett 'utvecklarcertifikat', det vill säga ett certifikat som endast är giltigt på localhost. Däremot så skedde någonting någongång som gjorde att det slutade fungera. Konsekvensen av detta blev att webbläsaren blockerade alla anslutningar till och från API:et, och hela webbapplikationen slutade fungera.

Detta problemet tros ha uppstått när en teammedlem tog bort package.json och node_modules-mappen, för att sedan återställa dessa igen med kommandot `npm i`. Information om något visst paket måste ha funnits enbart i package.json, och sedan då inte installerats igen. Lösningen på problemet var helt enkelt att skapa ett nytt projekt med hjälp av kommandot `dotnet new react -o [namn på projekt]`, och sedan ersätta package.json samt package-lock.json i vårt projekt med de som skapades i det nya projektet. Sedan så ominstallerades alla paket vi lagt till själva manuellt. När detta var gjort skapades ett giltigt certifikat och anslutningar till och från API:et blockerades inte längre.

Skulle man vilja sätta ut applikationen på riktigt, så att den görs tillgänglig över internet så måste givetvis ett riktigt certifikat genereras, något som experimenterades med och faktiskt lyckades, iallafall för API:et. Det var helt enkelt att lägga till ett NuGet-paket som heter *LettuceEncrypt* och göra minimal konfiguration. Processen över hur det görs finns väl beskrivet i README-sidan hos paketets NuGet-sida (Microsoft Nuget, 2022). Sedan krävdes även lite konfiguration av domänen, där en av teammedlemmarna redan hade en domän. Där las bara en underdomän till som pekade på API:et på en teammedlems egna dator hemma. Tanken med detta var att alla teammedlemmar kunde arbeta mot samma databas. Däremot så var API:et och databasen under konstant utveckling, så detta visade sig i slutändan ha liten praktisk nytta. Det finns även säkerhetsrisker med att öppna upp API:et till hela internet, och vi vet om att det finns säkerhetshål i vårt API. Konsekvenserna av en attack i just vårt fall är relativt begränsade, i och med att

databasen är fylld med påhittad data. Det enda som skulle vara känsligt är att en profils GoogleId görs tillgängligt genom API:et, och i och med att vi testar att logga in med våra riktiga Google-konton så skulle det kunna vara känslig information. Däremot så visade det sig att GoogleId inte gav någon större tillgång till data, utan bara information som redan är publikt hos ett Google-konto.

Ett annat problem vi hade var med inladdningen av data till React-appen. I React så finns det inget pålitligt sätt att exekvera kod före en komponents DOM-element har renderats ut på sidan. Man kan däremot exekvera kod direkt efter att en komponents DOM-element renderats ut (on mount) och efter en komponent (och motsvarande DOM-element) tagits bort (unmount). Detta görs med hjälp av useEffect-hooken. Problemet vi hade var att data hämtades från API:et, efter en komponents monterats. Det vill säga, datat som behövdes för att montera komponenten fanns inte ännu. Det orsakade en hel del fel när komponenten försökte renderas. Lösningen var att hålla koll på om datat har hämtats än eller inte, och om det inte har det så renderas en laddningsanimation istället. Det vill säga att vi såg till att variabler inte avläses innan de finns.

Ett eventuellt problem vi kan ha, är att vi inte är helt säkra på om vi implementerat Google-login med OAuth på ett korrekt sätt. Vad detta kan innebära är en säkerhetsrisk om vi gjort det på ett felaktigt sätt. Av denna anledning skulle det behöva undersökas ytterligare vid en vidareutveckling.

7.2 Reflektioner

Vi i projektgruppen är överens om att det varit ett mycket intressant och givande projekt. Vi är också tacksamma att det gick att slå ihop projektmomenten i de båda kurserna eftersom vi hann implementera mycket mer och bredare funktionalitet än om vi skulle gjort två separata projekt. Den största lärdomen är därför att ha gjort ett så pass stort projekt från start till mål, där omfattande backend- och frontendimplementation och logik skulle vävas samman till ett komplett och fungerande system.

Vi är också överens om att teknikutvärderingen som gjordes innan projektet var till stor hjälp för att komma igång med arbetet. Vi fick redan då goda insikter om vad vi ville göra och hur detta skulle genomföras. Det möjliggjorde även att vi tidigt kunde strukturera projektet på ett överskådligt sätt, eftersom vi i förväg kunde tänka ut de olika momenten som skulle ingå.

8 GitHub-repo

Länk till vårt Git-repo (eller, https://github.com/emmylindgren/AFI-Project)

Referenser

Freaky Jolly. (2022). *React Google SignIn/ Login Button Example using React-Google_ Login Package*. https://www.freakyjolly.com/google-signin-login-button-in-react-js-example-using-react-google_login-package/ (accessed: 26.05.2022)

Google Identity. (2022). *Authenticate with a backend server*. https://developers.google.com/identity/sign-in/web/backend-auth (accessed: 25.05.2022)

Microsoft. (2022). *dotnet-aspnet-codegenerator*. https://docs.microsoft.com/en-us/aspnet/core/fundamentals/tools/dotnet-aspnet-codegenerator?view=aspnetcore-6.0 (accessed: 26.05.2022)

Microsoft Nuget. (2022). *LettuceEncrypt*. https://www.nuget.org/packages/LettuceEncrypt/ (accessed: 26.05.2022)

React. (2022a). *Lifting State Up*. https://reactjs.org/docs/lifting-state-up.html (accessed: 26.05.2022)

React. (2022b). *Main Concepts*. https://reactjs.org/docs/hello-world.html (accessed: 25.05.2022)

React Router. (2022). *Docs*. https://reactrouter.com/docs/en/v6 (accessed: 26.05.2022)