

# Fixmeapp

Skapandet av databas och admingränssnitt

Aleksandra Kuzeleva

**MITTUNIVERSITETET**  
**Institutionen för data- och elektroteknik (DET)**

**Examinator:** Lars Lundin, [lasr.lundin@miun.se](mailto:lasr.lundin@miun.se)  
**Handledare:** Mikael Hasselmalm, [mikael.hasselmalm@miun.se](mailto:mikael.hasselmalm@miun.se)  
**Författare:** Aleksandra Kuzeleva, [alku2200@student.miun.se](mailto:alku2200@student.miun.se)  
**Utbildningsprogram:** TWEUG, Webbutveckling, 120 hp  
**Huvudområde:** Datateknik  
**Termin, år:** VT, 2024

# Sammanfattning

**Nyckelord:** Människa-dator-interaktion, XML, Linux , Java.

## Abstract

Abstract, det vill säga motsvarande sammanfattning på engelska, krävs i examensrapporter. Abstract skrivs i ett stycke.

**Keywords:** Exempel: Human-computer interaction, XML, Linux, Java.

## Förord

Förord är inte obligatoriskt men kan tillämpas om du som skribent vill inkludera några personliga ord, till exempel tack till personer som hjälpt dig. Denna text ska alltid skrivas på en egen sida.

# Innehållsförteckning

<b>Sammanfattning.....</b>	<b>2</b>
<b>Abstract.....</b>	<b>3</b>
<b>Förord.....</b>	<b>4</b>
<b>Terminologi.....</b>	<b>7</b>
<b>1    Introduktion.....</b>	<b>1</b>
1.1    Bakgrund och problemmotivering.....	1
1.2    Övergripande syfte.....	1
1.3    Avgränsningar.....	2
1.4    Detaljerad problemformulering.....	2
1.4.1    Målformuleringar.....	2
1.5    Översikt.....	3
<b>2    Teori.....</b>	<b>4</b>
2.1    SQL.....	4
2.2    Node.js och Express.js.....	4
2.3    API och REST API.....	4
2.4    API key.....	5
2.5    SendGrid.....	5
2.6    GitHub versionshantering.....	5
2.7    uuid.....	5
2.8    bcrypt.....	6
2.9    React.....	6
2.10    Axios.....	6
2.11    Bootstrap.....	7
<b>3    Metod.....</b>	<b>8</b>
3.1    Planering.....	8
3.2    Databas.....	8
3.3    Utvecklingen av programkod.....	8
3.4    Versionshantering.....	9
3.5    Möten.....	9
3.6    Tester.....	9
<b>4    Konstruktion.....</b>	<b>10</b>
4.1    Databas.....	10
4.1.1    Users och services.....	11
4.1.2    Servicerekommendationer.....	11
4.1.3    Administratörer.....	12
4.2    Back-end.....	13
4.2.1    Att sätta upp utvecklingsmiljön.....	13
4.2.2    Databasinitiering.....	13
4.2.3    API översikt.....	16
4.2.3.1    Skapa (Create).....	16

4.2.3.2	Läsa (Read).....	16
4.2.3.3	Uppdatera (Update).....	17
4.2.3.4	Radera (Delete).....	17
4.2.3.5	Ytterligare operationer.....	17
4.2.4	Registrering av ny administratör.....	18
4.2.5	Registrering av inloggningsförsök.....	21
4.2.6	Cookies.....	22
4.2.7	Testning.....	23
4.3	Front-end.....	23
4.3.1	Routing.....	24
4.3.2	Komponentarkitektur.....	25
4.3.3	Komponenter.....	25
4.3.4	Sidor.....	27
4.3.4.1	Inloggninssida.....	27
4.3.4.2	Sidan för lösenordsändring.....	28
4.3.4.3	Dashboard.....	30
4.3.4.4	Sidan admins.....	31
4.3.5	Testning.....	32
<b>5</b>	<b>Resultat.....</b>	<b>33</b>
<b>6</b>	<b>Slutsatser.....</b>	<b>35</b>
6.1	Etisk och social diskussion.....	35
	<b>Källförteckning.....</b>	<b>37</b>

## Förkortningar

SQL	Structured query language
ER-diagram	Entity-relationship diagram
REST	Representational State Transfer
API	Application Programming Interface

# 1 Introduktion

## 1.1 Bakgrund och problemmotivering

Fixmeapp AB är en startup som har som mål att skapa en applikation för bokning av olika tjänster inom skönhetsbranschen: klippning, kosmetiska ingrepp, massage etc. Applikationen kommer att kunna genom att ha tillgång till användares och tjänsteleverantörers kalendrar, samt till deras platser, snabbt hitta lämplig tid. Applikationen bör också kunna ta hänsyn till användarens hårtyp när de rekommenderar en frisör. Fixmeapp AB är en startup som har som mål att skapa en applikation för bokning av olika tjänster inom skönhetsbranschen: klippning, kosmetiska ingrepp, massage etc. Applikationen kommer att kunna genom att ha tillgång till användares och tjänsteleverantörers kalendrar, samt till deras platser, snabbt hitta lämplig tid. Applikationen bör också kunna ta hänsyn till användarens hårtyp när de rekommenderar en frisör. Denna funktion kan implementeras med AI-teknik eller möjligen andra metoder. Under detta projekt skapas en bas för den framtida applikationen: databas och en del av REST API som sedan kommer att konsumeras av applikationen.

Fixmeapp kan bli ett praktiskt verktyg för att boka skönhetsjänster som kan spara mycket tid för användarna. Dessutom kan det bli ett användbart instrument för tjänsteleverantörer som vill marknadsföra sina tjänster och hitta nya kunder. Det är viktigt för både de tjänsteleverantörer som redan är etablerade på marknaden och de som precis börjat skapa sin kundbas. Bland annat Fixmeapp kan användas av tjänsteleverantörer som inte talar svenska (eller kanske också är engelska) och har svårt att hitta kunder. Applikation kan hjälpa tjänsteleverantörer som kom från till exempel Ukraina eller Syrien att ta sina första steg på den svenska marknaden och hjälpa dem att kunna försörja sig själva.

## 1.2 Övergripande syfte

Det övergripande syftet med projektet Fixmeapp AB är att förbättra interaktionen mellan kunder och tjänsteleverantörer inom skönhetsindustrin genom att använda digital teknologi i bokningsprocessen. Vi strävar efter att skapa en ny användarupplevelse där vi använder artificiell intelligens och dataanalys för att anpassa tjänster efter individuella behov och preferenser, vilket förhoppningsvis både förbättrar kundtillfredsställelsen och effektivisera verksamheten för tjänsteleverantörerna. Bolaget strävar efter att skapa en plattform som hanterar och skyddar användardata samtidigt som den erbjuder personliga anpassade tjänster.



## 1.3 Avgränsningar

I detta projekt kommer fokus att ligga på utvecklingen av de tekniska grundkomponenterna för Fixmeapp, medan integreringen av avancerade AI-funktioner och mer komplexa dataanalysmetoder kommer att hanteras i framtida utvecklingsfaser. Viktigt att notera att projektet är skapat från grunden, så ingen tidigare skapad källkod fanns tillgänglig.

## 1.4 Detaljerad problemformulering

I detta avsnitt specificeras de konkreta mål som projektet strävar efter att uppnå. Målformuleringen är direkt kopplad till de övergripande syftena och inriktar sig på att leverera mätbara och verifikationsbara resultat som kan användas för att utvärdera projektets framgång vid dess avslutning.

### 1.4.1 Målformuleringar

#### Databas:

- Utveckla en databas som effektivt lagrar och hanterar kund- och administratörsdata.
- Implementera säkerhetsåtgärder som spårar användarnas senaste inloggningstider och registrerar misslyckade inloggningsförsök.

#### REST API

- Skapa ett REST API för hantering av användare och administratörer inklusive autentisering.
- Möjliggöra administrativa åtgärder såsom inaktivering och borttagning av användarkonton.
- Registrering av användare och administratörer bör vara fullständig efter bekräftelse via e-post.

#### Front-end

- Skapa admin-gränssnitt där man kan registrera nya administratörer, se och redigera data för administratörer och användare, se lista över misslyckade inloggningsförsök, se och redigera sin egen profil.
- Administratörsgränssnittet ska endast vara tillgängligt för inloggade användare med status "active" i databasen.

- På inloggningssidan bör det finnas länken "Glömt lösenord" genom att klicka på vilken användare som omdirigeras till en sida där man kan återställa lösenordet.
- Inloggade användare ska också kunna ändra lösenord på sina profilsidor.

## 1.5 Översikt

Kapitel 1 beskriver projektets bakgrund, avgränsningar samt målformuleringar.

Kapitel 2 innehåller en lista över teoretiska termer som är viktiga för att förstå rapporten.

Kapitel 3 beskriver metoder som användes i projektarbetet.

Kapitel 4 handlar om konstruktion av databasen, back-end och front-end.

I kapitel 5 sammanfattas projektets resultat.

Kapitel 6 contains slutsatser samt etisk diskussion.

## 2 Teori

I detta avsnitt förklaras de termer som är viktiga för att förstå rapporten. Det förutsätts att läsarna är bekanta med de mest grundläggande termerna inom webbutveckling, såsom HTML, CSS, databaser, JavaScript, ramverk, back-end och front-end.

### 2.1 SQL

SQL (Structured Query Language), är ett programmeringsspråk som används för att skapa relationsdatabaser och hantera data som lagras där. Relationsdatabaser lagrar data i tabeller, som består av rader och kolumner, vilket möjliggör effektiv datalagring och relationskartläggning mellan olika dataenheter.(1)

SQL utvecklades på 1970-talet av Edgar F. Codd och då hade den namnet SEQUEL (Structured English Query Language) på grund av att dess syntax mycket liknar engelska. SQL-databaser kan integreras med många olika ramverk, varav en är Node.js.(1)

### 2.2 Node.js och Express.js

Node.js är en öppen källkod, plattformsoberoende miljö som tillåter använda JavaScript-kod på serversidan. Ursprungligen begränsades JavaScript till operationer på klientsidan i webbläsare. Med Node.js har det blivit möjligt att även använda JavaScript för back-end utveckling. Att använda samma språk på både front-end och back-end förenklar utvecklingsprocessen. (8)

Express.js är ett ramverk för Node.js som förenklar skapandet av webbapplikationer och API:er. Det finns många olika bibliotek och plugins som kan användas med Express.js och som gör det möjligt att skapa applikationer av varierande funktionalitet och komplexitet. (8)

### 2.3 API och REST API

API eller Application Programming Interface, är en viktig verktyg som tillåter olika programvarusystem att kommunicera med varandra med hjälp av en uppsättning regler och protokoll. Ett sätt att förstå vad är API är att se det som en kontrakt mellan två applikationer. I kontraktet beskrivs det korrekta sättet eller regler för interaktion mellan applikationerna. (2)

En specifik typ av API – REST (Representational State Transfer) API, använder principer för representativ tillståndsoverföring för att förbättra skalbarhet och flexibilitet i kommunikation över internet. REST API:er använder HTTP (Hypertext Transfer Protocol) - protokoll som driver webben, vilket gör dem särskilt användbara för webbinteraktioner. (2)

## 2.4 API key

En API key är en unik identifierare vilken fungerar som både en identifierare och en autentiseringsmekanism för applikationer och användare som interagerar med API. Dessa nycklar är viktiga för att hantera interaktionen mellan programvarukomponenter: de hjälper till att säkerställa att förfrågningar till API:er görs säkert och av auktoriserade enheter.(3)

## 2.5 SendGrid

För att förklara vad är SendGrid är det viktigt att kort beskriva processen bakom e-postfunktionalitet. När användare klickar på Skicka-knappen börjar e-postmeddelanden sin resa via Simple Mail Transfer Protocol (SMTP), som ansvarar för att verifiera och lagra dessa meddelanden. När ett e-postmeddelande är avsett för en annan domän, samordnar SMTP med mottagarens domännamnsserver (DNS) för att säkerställa leverans, hantera potentiella problem genom att köa eller returnera meddelanden som inte kan levereras.(9)

SendGrid är en av verktygen som kan användas för att hantera processen för att skicka e-postmeddelanden med en app. Det är en molnbaserad SMTP-tjänst passande för att hantera stora e-postvolymmer. (9)

## 2.6 GitHub versionshantering

Versionskontroll tillåter att ha koll över ändringar som gjorts i programkod och om det behövs gå tillbaka till några av de tidigare versionerna av koden. (11)

Git är en populär versionskontrollmjukvara som underlättar spårning av ändringar i programkod. Git i kombination med GitHub tillåter både versionshantera projekt och samarbeta med andra utvecklare. (11)

## 2.7 UUID

En UUID (Universally Unique Identifier) är ett 128-bitars nummer som används för att unikt identifiera delar av information i datorsystem.

Inom mjukvaruutveckling, till exempel i miljöer som använder Node.js, är UUID:er avgörande för att upprätthålla dataunikitet och förhindra fel. I Node.js kan UUID genereras med olika paket, till exempel paketet uuid. (12)

## 2.8 bcrypt

bcrypt är en kryptografisk hashfunktion utvecklad för att förbättra lösenordssäkerheten. Bibliotek bcrypt.js i Node.js gör det möjligt att enkelt hasha lösenord och därmed avsevärt förbättra applikationens säkerhet. (10)

## 2.9 React

React.js, vanligen förkortat som React, är ett JavaScript-bibliotek som har blivit en populär verktyg inom webbutveckling. React-webbapplikationen är sammansatt av återanvändbara komponenter som utgör delar av användargränssnittet — det kan finnas en separat komponent för navigeringsfältet, för sidfoten, en annan för huvudinnehållet och så vidare. Dessa återanvändbara komponenter gör utvecklingen lättare eftersom det inte är nödvändigt att upprepa återkommande kod i olika filer. Komponent skapas en gång och importeras sedan till valfri del av koden där den behövs. På så sätt kan vissa komponenter användas i flera applikationer. (7)

Syntax som används för att bygga React-appar kallas JSX (JavaScript XML), som är ett syntaxtillägg till JavaScript. (7) JSX ser ut som en blandning av JavaScript och HTML-taggar och det eliminerar behovet av att använda till exempel `document.getElementById`, `querySelector` och andra metoder som används i JavaScript för att manipulera DOM. (7)

## 2.10 Axios

Axios är ett populärt JavaScript-bibliotek som används för att göra HTTP-förfrågningar från en webbläsare eller Node.js-server till externa resurser, såsom API:er. Bibliotekets promise-baserade karaktär gör att det effektivt kan hantera asynkrona förfrågningar (requests) ett centralt krav i moderna webbapplikationsarkitekturer. Det ger en mer läsbar syntax jämfört med mer traditionella metoder (till exempel callback-metoder).

När Axios väl har integrerats i ett projekt kan den användas för att göra olika typer av HTTP-förfrågningar, till exempel de grundläggande CRUD-operationerna (CREATE, READ, UPDATE, DELETE). Axios hanterar dessa med sina metoder som `axios.get()`, `axios.post()`, `axios.put()` och `axios.delete()`. (4)

## 2.11 Bootstrap

Bootstrap är ett ramverk med öppen källkod som avsevärt förenklar utvecklingen av responsiva webbsidor. En av nyckelfunktionerna i Bootstrap är dess rutsystem, som gör det möjligt för utvecklare att skapa adaptiva och skalbara layouter som fungerar på på de flesta enheter. Detta rutsystem kompletteras med en omfattande samling verktyg för hantering av typografi, formulär, knappar, bilder och mer. Bootstrap är kompatibel med många front-end ramverk, bland annat React. (5)

## 3 Metod

### 3.1 Planering

Eftersom applikationen skapas från grunden var planering en mycket viktig del av projektarbetet, även om det redan från början stod klart att projektet och planen kommer att förändras många gånger. Tidsplanen skapades som ett enkelt Gantt-diagram. Ett onlineverktyg Trello användes för att planera och följa upp arbetet med projektet.

### 3.2 Databas

Skapandet av databasschemat och motsvarande Entity-Relationship (ER) diagram gjordes i nära samarbete med en mentor på företaget. Detta steg var avgörande för att säkerställa att databasstrukturen effektivt stöder applikationens databehov och är i linje med applikationslogiken. Första steget var att diskutera applikationen i alla detaljer för att avgöra vilken data som måste lagras i databasen och hur användare och administratörer ska kunna manipulera denna data. Prototypen av mobilapplikationen användes för denna analys.

Därefter diskuterades typen av databas och handledaren föredrog SQL då hon var mer bekant med den. Efter det skapades SQL-koden och databasens ER-diagram. Verktyget dbdiagram.io användes för att skapa ER-diagram. dbdiagram.io gör det möjligt att skapa ER-diagram baserat på tillhandahållen (någorlunda modifierad) SQL-kod, vilket avsevärt minskar tiden för att skapa diagrammet. Databasen testades sedan med en rad SQL-frågor.

### 3.3 Utvecklingen av programkod

All kod skapades i Visual Studio Code. Back-end skapades med Express.js, front-end med React.

Express.js valdes för backend eftersom det är reativt lätt att använda och för att det tillåter att ganska snabbt bygga REST API: er. Dess stöd för middleware förbättrar funktionaliteten för viktiga uppgifter som loggning av förfrågningar och användarautentisering, medan dess routingfunktioner underlättar hanteringen av RESTful API. Den omfattande dokumentationen, stora mängder artiklar och tutorials om Express.js gör det lättare att lösa problem som uppstår under utvecklingsprocessen.

För frontend valdes React främst för sin komponentbaserade arkitektur som uppmuntrar modularitet och återanvändbarhet, avgörande för att underhålla och skala applikationer. Möjligheten att återanvända några av komponenterna gör utvecklingsprocessen snabbare och enklare. Den sista men mycket viktiga fak-

torn som övervägdes är att handledaren då ännu inte bestämt var webbappen ska installeras. De mest troliga valen var Google Cloud eller Amazon Web Services och båda tillåter deployment av applikationer som använder MariaDB, Express.js och React.

Huvudfokus under projektet var databasen och back-end så designen av admin gränssnittet var tänkt att vara enkelt, skapad med hjälp av Bootstrap. Det antas att det kommer att utvecklas vidare i framtiden och kommer att få design som är i linje huvudapplikationens design.

### **3.4 Versionshantering**

Programkod under hela projektarbetet versionshanterades i GitHub. Mentor och utvecklaren i teamet har tillgång till repon med programkod.

### **3.5 Möten**

I de inledande skedena av projektarbetet, medan databasen och hela applikationen planerades, hölls dagliga möten med mentor där alla projektdetaljer diskuterades. På de fortsatta stadierna av projektarbetet var det dagliga korta stand-ups där projektet följdes upp.

### **3.6 Tester**

REST API:et testades med Thunder Client och Postman - verktyg för HTTP-klienttestning. Dessa verktyg underlättade simuleringen av klientförfrågningar till backend, vilket gjorde det möjligt att noggrant inspektera API-svar, hantera olika API-slutpunkter och felsöka. Detta tillvägagångssätt hjälpte till att validera funktionaliteten hos vår Express.js-backend och säkerställa att API:et uppfyllde de specificerade kraven.

För frontend användes W3C Validator för att säkerställa att HTML och CSS uppfyllde webbstandarder och var fria från fel. Denna valideringsprocess var avgörande för att upprätthålla standarder för webbkompatibilitet och prestanda.

## 4 Konstruktion

### 4.1 Databas

Utvecklingen av applikationen började från grunden. I utvecklingsprocessen utgick jag från prototypen av mobil applikation skapad i Figma och idéer från min handledare på företaget som kom på idén att skapa Fixmeapp. Eftersom detta projekt är begränsat till att skapa databas, back-end och admin-gränssnitt är prototypen irrelevant för rapporten. Det användes dock för att analysera vilka data som måste lagras i databasen och hur dessa data ska kopplas samman. Enligt denna analys bör databasen lagra data om:

- Användare;
- Administratörer;
- Bokningar;
- Tjänster och deras kategorier;
- Priser.

Senare beslutades att även lagra data om administratörers och användares aktiviteter i applikationen (till exempel senaste inloggningstid och misslyckade inloggningsförsök), betyg från användare, användares vänner (i den första, begränsade, versionen av applikationen och kommer att utvecklas senare).

Under de första två veckorna av projektet skapades databasen med tabeller admins (för adminsdata), admins inloggningsförsök (för admins inloggningsförsök som kommer att lagras i databasen), användare (för kunddata), tjänsteleverantörer (för frisörer, makeupartister etc), tjänster och bokningar (bokningsinformation). Sedan började utvecklingen av backend och frontend för admin-gränssnitt, vilket beskrivs i respektive avsnitt. Samtidigt fortsatte diskussionen om applikationens funktionalitet och databas med mentorn och efter en tid gjordes flera betydande förändringar i databasen. Till exempel togs beslutet att lagra kunder och tjänsteleverantörer till en enda tabell Users. Detta tillvägagångssätt tillåter individer att använda applikationen som både kunder och tjänsteleverantörer med ett enda konto, vilket eliminerar besväret med att ha två konton. Det minskar också dataredundans och förenklar underhållet. Att använda en tabell istället för separata för olika användarroller effektiviserar utvecklingen, förbättrar databehandlingshastigheterna och säkerställer konsekvens över hela plattformen, vilket gör systemet både skalbart och lättare att hantera.

Vidareutvecklingen av databasen fortsatte under hela projektarbetet. Nu innehåller databasen tjugoen tabell och det är mest troligt att den kommer att ut-



vecklas vidare. På företagets begäran kan databasens hela ER-diagram inte publiceras, men alla databasens tabeller är listade i kodbilden på Figur 4.

#### 4.1.1 Users och services

Som nämnts ovan togs beslutet att lagra data som gäller både kunder och tjänsteleverantörer i en gemensam tabell, Users. Detta skulle möjliggöra för en person att använda applikationen både som kund och som tjänsteleverantör med ett enda konto, istället för att skapa två separata konton. Tabellen Services länkar tjänster till de användare som erbjuder någon form av tjänster, till exempel frisyr eller manikyr. Koden för dessa två tabeller visas i figur 1.

```
// Users Table Schema
const usersTableSchema = `
CREATE TABLE IF NOT EXISTS users (
  id INT AUTO_INCREMENT PRIMARY KEY,
  username VARCHAR(255) UNIQUE NOT NULL,
  email VARCHAR(255) UNIQUE NOT NULL,
  passwordHash VARCHAR(255) NOT NULL,
  firstName VARCHAR(255) NOT NULL,
  lastName VARCHAR(255) NOT NULL,
  phoneNumber VARCHAR(20),
  image VARCHAR(255),
  status ENUM('active', 'inactive', 'suspended'),
  createdAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  language VARCHAR(255),
  timeZone VARCHAR(255)
);

const servicesTableSchema = `
CREATE TABLE IF NOT EXISTS services (
  id INT AUTO_INCREMENT PRIMARY KEY,
  providerID INT,
  categoryID INT,
  serviceName VARCHAR(255) NOT NULL,
  description TEXT,
  priceID INT,
  isActive BOOLEAN NOT NULL DEFAULT 1,
  FOREIGN KEY (providerID) REFERENCES users(id),
  FOREIGN KEY (categoryID) REFERENCES categories(id),
  FOREIGN KEY (priceID) REFERENCES prices(id)
);
```

Figur 1

Användare som bara använder applikationen som kunder är inte länkade till tabellen Services.

#### 4.1.2 Servicerekommendationer

Det är planerat att i framtiden ska Fixmeapp kunna rekommendera frisörer (och senare andra tjänsteleverantörer, men den första, begränsade, versionen av applikationen fokuserar på klippningstjänster) utifrån flera faktorer. Hårtyp (långt eller kort, tjockt eller tunt, lockigt eller rakt hår etc) kan påverka priset och hur lång tid klippningen kan ta. Dessutom kan tjänstens varaktighet variera från fri-

sör till frisör. För att lagra data om hårtyper och tjänstens varaktighet, och för att sedan i appen kunna visa förväntad servicelängd för klienter, skapades hår-typstabeller, hårtyper, varaktighet och servicelängd. Tabel Durations innehåller standardvaraktigheter, som i tabellen ServiceDurations är kopplade till olika tjänster. Alternativt kan varje frisör ha en varaktighet som är annorlunda än standarden. Tabell servicePriceAdjustments kommer att användas för att justera pris baserat på tjänstetyp, frisörernas tillgänglighet och kundens lojalitet.

#### 4.1.3 Administratörer

Tabellen Admins är skapad för att lagra omfattande information om varje administratör. Detta inkluderar namn och kontaktuppgifter, användarnamn (det genereras automatiskt vilket beskrivs i backend avsnittet), unika e-postadresser och hashade lösenord. Varje administratör har också en viss roll inom organisationen: "superadmin", "admin" eller "support". Det kommer att användas för att skapa funktionalitet kring olika åtkomst- och kontrollnivåer inom applikationen i senare utvecklingsfaser.

Administratördata inkluderar tvåfaktorsautentiseringsflaggor (twoFactorEnabled), multifaktorausautentiseringsaktiverad status (MFAEnabled) och fält för lagring av den senaste inloggnings-IP-adressen (lastLoginip) för att övervaka åtkomstmönster. För närvarande implementeras tvåfaktorausautenticeringen i form av registreringsbekräftelse via e-post. Multifaktorausautentisering har ännu inte implementerats.

Unika fält för apiKey och token ingår också, vilka är viktiga för API-åtkomst och sessionshantering, vilket förbättrar både säkerheten och mångsidigheten för administratörsinteraktioner med systemet.

Tabellen adminLoginAttempts registrerar varje inloggningsförsök av en administratör, inklusive tidsstämpeln och IP-adressen från vilken försöket gjordes. Detta är viktigt för säkerhetsgranskning och för att identifiera potentiella obehöriga åtkomstförsök. Det säkerställer att det finns en spårbar logg över vem som gick åt administratörspanelen och när, vilket är grundläggande för att upprätthålla integriteten för systemdriften.

För varje administratör lagras tidsstämplar för kontoskapande (createdAt) och senaste uppdatering (updatedAt), som hanteras automatiskt av databassystemet. I fältet lastLogin registreras tidpunkten för den senaste lyckade inloggningen.

## 4.2 Back-end

I detta avsnitt beskrivs konstruktionen av back-end som konsumeras av admingränssnittet och den färdiga delen av back-end som sedan kommer att konsumeras av användargränssnittet.

### 4.2.1 Databasinitiering

Databasinitieringen och konfigurationen hanteras av koden i filen `database.js`. Den här koden är utformad för att köras automatiskt när projektet startar, exekvera SQL-kommandon som skapar nödvändiga databastabeller om de inte redan finns och upprättar relationer enligt definitionen i applikationens datamodell. Anslutningsinställningarna, såsom databasserveradressen och andra parametrar, anges i en separat konfigurationsfil (`config.js`), som `database.js` läser under körningen. Denna process säkerställer att databasschemat är korrekt inställt varje gång programmet distribueras eller startas om, vilket gör backend-koden direkt ansvarig för att databasen skapas.

```
const mysql = require('mysql');
const config = require('./config');

const pool = mysql.createPool(config.database);
```

Figur 2

Figur 2 visar att koden börjar med att importera MySQL-biblioteket (som gör att Node.js-applikationer kan utföra uppgifter som att ansluta till en MySQL-databas, exekvera SQL-frågor och hantera databasoperationer asynkront) och en lokal konfigurationsmodul (som innehåller databaskonfiguration). Denna inställning tillåter servern att interagera med MySQL-databaser och använda serverinställningar definierade i filen `config.js`. Sedan skapas MySQL-anslutningspoolen med hjälp av databasinställningarna från `config.js`.

Anslutningspoolen är viktig eftersom den sparar tid och resurser. Istället för att öppna en ny anslutning varje gång applikationen behöver kommunicera med databasen, håller poolen redan öppna anslutningar redo att användas när som helst. När applikationen behöver ansluta till databasen, plockar den en av de redan öppna anslutningarna från poolen, använder den och sedan lägger tillbaka den när den är klar. Det gör att applikationen kan kommunicera med databasen snabbare och hantera fler saker samtidigt, vilket gör hela processen snabbare och mer effektiv.

Koden i `database.js` innehåller även definitioner för olika tabellscheman. Dessa definierar strukturen för databastabellerna, inklusive primärnycklar, främmande nycklar, datatyper mm. Exempel på ett sådant tabellschema visas på figur 3.

```
const adminLoginAttemptsTableSchema = `
CREATE TABLE IF NOT EXISTS adminLoginAttempts (
  attemptID INT AUTO_INCREMENT PRIMARY KEY,
  adminID INT NOT NULL,
  attemptTime TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  ipAddress VARCHAR(45) NOT NULL,
  FOREIGN KEY (adminID) REFERENCES admins(id)
);
`;
```

Figur 3

Eftersom databasen innehåller nu många tabeller, varav de flesta har minst en främmande nyckel, kan användning av ett sådant skript som i database.js för att skapa alla dessa tabeller orsaka främmande nyckelrelaterade fel (till exempel foreign key constraint is incorrectly formed). För att undvika detta används följande kod:

```
const createTables = async () => {
  const baseTables = [
    usersTableSchema, // Users table must exist before any other references
    categoriesTableSchema, // Categories must exist before services
    createDurationTable,
    createHairTypeTable,
    servicesTableSchema, // Services must exist before ratings, bookings, user history
    adminsTableSchema,
    pricesTableSchema,
  ];

  const dependentTables = [
    userActivityLogTableSchema, // Depends on users
    userLocationTableSchema, // Depends on users
    userFriendsTableSchema, // Depends on users
    privacySettingsTableSchema, // Depends on users
    userPreferencesTableSchema, // Depends on users
    createServiceDurationTable, // Depends on services
    bookingsTableSchema, // Depends on users and services
    userHistoryTableSchema, // Depends on users and services
    ratingsTableSchema, // Depends on users and services
    adminLoginAttemptsTableSchema, // Depends on admins
    availabilityTableSchema,
    userLoginAttemptsTableSchema,
    loyaltyDiscountsTableSchema,
    servicePriceAdjustmentsTableSchema,
  ];

  const schemas = [...baseTables, ...dependentTables];

  for (const schema of schemas) {
    pool.query(schema, (err, results) => {
      if (err) {
        console.error('Error creating table:', err.message);
        return;
      }
      console.log('Table created successfully');
    });
  }

  createTables();
  module.exports = { pool };
};
```

Figur 4

CreateTables-funktionen skapar en serie tabeller baserade på fördefinierade SQL-schemadefinitioner. Funktionen organiserar tabellscheman i två grupper:

**baseTables** - grundläggande tabeller som måste skapas först eftersom andra tabeller är beroende av dem.

**dependentTables** - tabeller som beror på förekomsten av en eller flera av bastabellerna. Till exempel beror bokningstabellen på både users- och services-tabellerna eftersom den refererar till enheter i båda. Försök att skapa bokningstabell före tabeller users och services kommer att resultera i fel.

Funktionen kombinerar sedan arrayer baseTables och dependentTables till en enda array som kallas **schema**.

Därefter itererar funktionen över arrayen schema och utför SQL-kommandon för att skapa respektive tabell i databasen i rätt ordning. Den använder pool.query för att skicka dessa kommandon till MySQL-databasen. Om SQL-kommandot körs framgångsrikt loggar det "Tabell skapad framgångsrikt" till konsolen. Om det uppstår ett fel under exekveringen av ett SQL-kommando, loggas ett felmeddelande och stoppar exekveringen av den aktuella iterationen. Funktionen fortsätter sedan till nästa schema i listan.

Efter att ha definierat funktionen anropas createTables() omedelbart för att köras när denna modul laddas. Detta innebär att processen för att skapa tabeller startar så snart backend-servern startar (comrad node app.js används för detta). Dessutom exporteras poolen från denna modul, vilket gör den tillgänglig för andra delar av applikationen att använda för att köra ytterligare SQL-frågor. Denna inställning gör det möjligt att snabbt skapa en databas när den flyttas. När nya tabeller läggs till i databasen är det viktigt att ta hänsyn till främmande nycklar och på så sätt placera tabellschemat i listan över beroendetabeller eller bastabeller.

## 4.2.2 API översikt

I detta avsnitt finns en översikt över endpoints och den tillhörande CRUD-funktionaliteten som tillhandahålls av REST API (det gäller funktioner som för närvarande finns och fungerar).

### 4.2.2.1 Skapa (Create)

**Administratörsregistrering (POST /admins/register):** Skapar en ny admin-profil, inklusive generering av unika identifierare som UUID för API-nycklar och sändning av bekräftelsemail.

**Användar- och administratörsinloggning (POST /users/login, POST /admins/login):** Inleder en session genom att validera användaruppgifter och logga framgångsrika och misslyckade försök.

**Bilduppladdning (POST /:id/upload för både administratörer och användare):** Tillåter uppladdning av bilder för att anpassa admin- eller användarprofiler.

**Initiering av lösenordsåterställning (POST /admins/forgot-password):** Genererar en token och skickar ett e-postmeddelande för att initiera lösenordsåterställningsprocessen.

#### 4.2.2.2 Läs (Read)

**Hämta alla administratörer (GET /admins), alla användare (GET /users):** Hämtar listor över alla administratörer eller användare.

**Hämta specifik administratör (GET /admins/:id), specifik användare (GET /users/:id):** Hämtar detaljerad information om en specifik administratör eller användare.

**Hämtning av supportadmin (GET /admins/support):** Hämtar information om administratörer som är utsedda som supportpersonal.

**Hämta inloggningsförsök (GET /admins/allLoginAttempts):** Tillhandahåller loggar över alla inloggningsförsök.

#### 4.2.2.3 Uppdatera (Update)

**Uppdatering av admin och användarprofil (PUT /admins/:id, PUT /users/:id):** Uppdaterar befintliga profiler med ny data, som namnändringar eller rolltilldelningar.

**Bilduppdatering (PUT /:id/image för både administratörer och användare):** Uppdaterar den befintliga profilbilden med en ny.

**Lösenordsåterställning slutförd (POST /admins/password-reset):** Uppdaterar administratörens lösenord efter att ha validerat återställningstoken.

#### 4.2.2.4 Radera (Delete)

**Radering av admin och användarprofil (DELETE /admins/:id, DELETE /users/:id):** Tar permanent bort en admin- eller användarprofil från databasen.

**Bildradering (DELETE /:id/image för både administratörer och användare):** Tar bort en profilbild från användar- eller adminprofilen och tar bort filen från servern.

#### 4.2.2.5 Ytterligare operationer

Dessa operationer passar inte exakt in i CRUD-modellen men är nödvändiga för full funktionalitet:

**Bekräftelse av administratörsregistrering (POST /admins/bekräfta-registrering):** Bekräftar administratörsregistreringen med en token, vilket ändrar status från inaktiv till aktiv.

**Hämta specifik administratör eller användarbild (GET /:id/image):** Skickar den faktiska bildfilen för en användare eller administratör, vilket möjliggör visuell representation i användargränssnittet.

Som nämnts ovan startas applikationens back-end med kommandon `node app.js`. I Express.js-applikationer fungerar filen `app.js` som den centrala filen där servern och dess primära konfigurationer ställs in. Det spelar en avgörande roll för att hantera interaktionen mellan inkommande förfrågningar, mellanprogram, routinglogik och svaret på klientsidan.

Koden i filen `app.js` initierar `express.js`-applikationen, importerar och använder modeller och kontroller för att hantera ruttspecifik logik. Modellerna i applikationen representerar datastrukturer och ansvarar för att interagera med databasen. De används av kontrollanter för att hämta, infoga, uppdatera eller radera data i databasen. I `Fixmeapp` interagerar `Admin` och `User` modeller dessutom med `utils` för att skicka e-post när ett nytt konto skapas (för bekräftelse) och när lösenordet ändras.

#### 4.2.3 Registrering av ny administratör

Processen för registrering av nya administratörer kan användas för att beskriva ovan nämnda API-inställning och interaktion mellan olika komponenter. Först, som vi redan vet, initieras applikationen i `app.js`. Det finns också importerade `body-parser` (mellanprogram för att analysera JSON-kroppar av inkommande förfrågningar), `bcrypt` (bibliotek som används för att hasha lösenord), sökväg (verktyg för att hantera och transformera filsökvägar), modeller och kontroller. För administratörsregistrering är `Admin`-modellen och `AdminController` nödvändiga. Förutom det `authenticateAdmin` - mellanprogram för att autentisera administratörer importeras också. Sedan kopplas controllers till specifika vägar för att organisera API:n och delegera förfrågningshantering till dessa styrenheter som det visas i figur 5.

```
// Mount controllers to specific routes
app.use('/users', usersController);
app.use('/admins', adminsController);

app.use('/images', express.static(path.join(__dirname, 'public/images')));
```

Figur 5

Här är `app.use('/admins', adminsController)` och `app.use('/users', usersController)` exempel på dynamisk innehållsdirigering där varje begäran till dessa sökvägar utlöser bearbetning i respektive styrenhet, potentiellt involverad databasoperationer och dynamiskt genererad svar. Å andra sidan hanterar `app.use('/images', express.static(path.join(__dirname, 'public/images')));`, statiskt innehåll. Servern svarar direkt med filer som finns i den angivna katalogen utan någon

bearbetning eller logikapplikation. Detta är optimalt för effektivitet och prestanda vid hantering av oföränderliga resurser som i det här fallet bilder.

För att gå tillbaka till administratörsregistrering skapas basökvägen `/admins` i `app.js`, medan de relativa sökvägarna - i `AdminController`. När klienten (admingränssnittet) skickar HTTP-begäran för att registrera en ny administratör är det POST-begäran till URL:en `.../admins/register` med nödvändiga uppgifter: förnamn, efternamn, e-post, telefonnummer och lösenord som ingår i förfrågans kropp. Begäran kommer till servern där `app.js` hanterar initial routing: den känner igen att begäranden endpoint börjar med `/admins` och vidarebefordrar begäran till `AdminController` på grund av ruttinställningarna `app.use('/admins', adminController)`.

När begäran har dirigerats till `AdminController`, identifierar den rätt hanterare för `/register`-sökvägen. Styrenheten validerar först inkommande data för att säkerställa att alla obligatoriska fält finns och är korrekt formaterade. Förutsatt att valideringen går igenom, fortsätter kontrollanten sedan att bearbeta data genom att interagera med administratörsmodellen för att skapa en ny administratörs-post.

Modellen `Admin` ansvarar för att interagera med databasen. Modellen skapar unika användarnamn, använder `uuid` för att generera API-nyckeln och infogar den nya administratörsdatan i databasen.

Figur 6 visar en del av funktionen `create` i `Admin`-modellen vilken genererar unika användarnamn:

```
create: async (adminData) => {
  try {
    // Generate unique username
    const firstNamePrefix = adminData.firstName.slice(0, 3).toLowerCase();
    const lastNamePrefix = adminData.lastName.slice(0, 3).toLowerCase();

    let username = `${firstNamePrefix}${lastNamePrefix}`;

    let isUnique = false;
    let suffix = 1;
    while (!isUnique) {
      const existingAdmin = await Admin.getByUsername(username);
      if (!existingAdmin) {
        isUnique = true;
      } else {
        suffix++;
        username = `${firstNamePrefix}${lastNamePrefix}${suffix}`;
      }
    }

    adminData.username = username;
  }
}
```

Figur 6

Funktionen extraherar först de tre första bokstäverna (prefix) i administratörens förnamn och efternamn och konverterar dem till gemener. Detta säkerställer att användarnamnet börjar med en igenkännbar del av administratörens faktiska



namn, vilket gör det något förutsägbart och lättare att komma ihåg. Dessa prefix kombineras sedan för att bilda den initiala basen för användarnamnet. Till exempel, om en administratörs namn är "Conny Newman", skulle användarnamnet vara connew. Det är dock möjligt att två eller flera administratörer eller användare kommer att ha samma för- och efternamn och användarnamn måste vara unika. För att säkerställa detta kontrolleras unikheten hos det nyligen genererade användarnamnet mot befintliga databasposter. Detta görs genom att fråga databasen för att se om någon befintlig administratör redan använder det genererade användarnamnet (metod `getByUsername` gör det). Om användarnamnet redan finns, läggs ett numeriskt suffix till i slutet, som börjar med 1. Systemet ökar detta suffix med ett för varje iteration tills ett unikt användarnamn hittas. Denna loop fortsätter och kontrollerar varje ny variant (`connew1`, `connew2`, etc.) som redan finns i databasen. När ett unikt användarnamn har identifierats tilldelas det till `adminData.username`. Detta säkerställer att varje administratör har ett distinkt användarnamn, vilket undviker potentiella konflikter och förvirring. Detta användarnamn används sedan för systeminloggningar, identifiering och eventuellt som en del av administratörens kontaktinformation inom organisationen.

Metoden **create** i modellen `Admin` hashar också lösenordet och genererar en token som kommer att användas för e-postverifiering. Efter att administratören har skapats har den statusen inaktiv tills registreringen har bekräftats via e-post.

`AdminsController` anropar e-postverktyget och ger det administratörens e-postadress och bekräftelsetoken. Token används för att säkerställa att e-postmeddelandet tillhör personen som registrerade sig.

Den mottagna token används för att bekräfta registreringen genom att kontrollera den mot databasposten med `Admin.confirmRegistration(token)`.

Om token är giltig och matchar, uppdateras administratörens status till aktiv, vilket innebär en framgångsrik registrering.

Ett 200 OK-svar skickas som indikerar framgångsrik registreringsbekräftelse om token matchar.

Om token inte matchar eller ett annat fel uppstår (t.ex. databassökning misslyckas) returneras lämpliga felmeddelanden med motsvarande HTTP-statuskoder (t.ex. 404 Not Found för en ogiltig token, 500 Internal Server Error för andra fel).

Figur 6 visar en del av `AdminController`-koden där ny admin skapas och status är inaktiv.

```
const result = await Admin.create({ ...adminData, apiKey, token, status: 'inactive' });
```

Figur 7

Den registrerade administratören får e-post med en bekräftelselänk. När den här länken klickas gör bekräftelsesidan på klientsidan POST-begäran med endpoint `admins/confirm-registration`. Den registrerade administratören får e-post med en bekräftelselänk. När den här länken klickas gör bekräftelsesidan på klientsidan en POST-begäran med slutpunktsadmin/bekräftelseregistrering och en token. Om token är densamma som token som sparats i databasen för den registrerade administratören, anses registreringen vara bekräftad och administratörens status ändras till aktiv.

Koden för registreringsbekräftelse i AdminController visas på figur 8.

```
// Confirm registration
router.post('/confirm-registration', async (req, res) => {
  const { token } = req.body;
  if (!token) {
    return res.status(400).json({ error: 'Token is required' });
  }
  try {
    const confirmationResult = await Admin.confirmRegistration(token);

    if (confirmationResult.error) {
      return res.status(404).json({ error: confirmationResult.error });
    }
    // Send a success response
    res.status(200).json({ message: 'Registration confirmed successfully' });
  } catch (error) {
    console.error('Error confirming registration:', error);
    return res.status(500).json({ error: 'Failed to confirm registration' });
  }
});
```

Figur 8

Denna kod kontrollerar först om token finns i begäran. Om inte, svarar den omedelbart med en 400 Bad Request-status, vilket indikerar att token krävs men inte tillhandahålls.

Om en token hittas anropar rutthanteraren metoden `confirmRegistration` för Admin-modellen och skickar token som ett argument. Där, om rätt token hittas, ändras administratörsstatusen till aktiv.

`confirmRegistration`-metoden i Admin-modellen returnerar ett resultatobjekt som antingen kan ha en `success`-egenskap (som indikerar en lyckad uppdatering) eller en `error`-egenskap (som indikerar att admin som matchar token inte hittades i databasen eller att ett annat fel uppstod).

Baserat på resultatet från metoden `confirmRegistration` bestäms svaret till klienten: om ett fel finns i resultatet skickar hanteraren ett 404 Not Found-svar med ett felmeddelande som indikerar att ingen motsvarande adminpost hittades för den angivna token. Om registreringen bekräftas framgångsrikt (d.v.s. databasen uppdaterades korrekt), skickas ett 200 OK-svar tillbaka till klienten med ett meddelande som säger "Registration confirmed successfully."

Användarregistreringen använder logik liknande den som beskrivs ovan.

#### 4.2.4 Registrering av inloggningsförsök

En viktig del av applikationssäkerheten är registrering av misslyckade inloggningsförsök. För att förklara hur detta fungerar kommer vi först att tala om inloggning (vi fortsätter att använda admins som exempel, men inloggningsprocessen för användare har samma logik). Basruttens administratör/inloggning som skapats i app.js lyssnar efter POST-förfrågningar och förväntar sig administratörsuppgifter i förfrågningstexten.

När begäran tas emot försöker systemet hämta en administratörspost med hjälp av den angivna e-postadressen eller användarnamnet (båda kan användas för att logga in) genom metoden `Admin.getByEmailOrUsername`. Om ingen administratör med angiven e-postadress eller användarnamn hittas, svarar den omedelbart med ett HTTP-svar 401 Ej auktoriserad status, vilket indikerar ogiltiga autentiseringsuppgifter.

Om en administratör hittas kontrolleras det angivna lösenordet mot det hashade lösenordet som lagras i databasen med hjälp av `bcrypt.compare`. Om lösenorden matchar får administratören tillgång till administratörsgränssnittet. Dessutom uppdaterar systemet den senaste inloggningstidsstämpeln i databasen för admin med `Admin.updateLastLogin`.

Om lösenordet inte matchar loggar systemet detta misslyckade försök genom att anropa metoden `Admin.registerUnsuccessfulLoginAttempt`. Denna metod tar ett `adminId` och en `ipAddress` och sparar försöket i tabellen `adminLoginAttempts` i databasen.

`req.ip` - property för objektet `req` (request) i Express.js används för att hämta fjärr-IP-adressen för begäran.

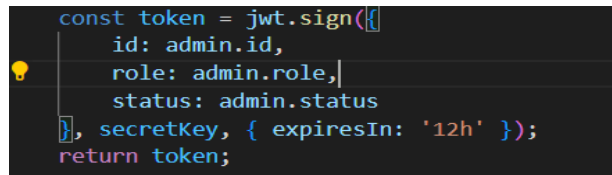
#### 4.2.5 Cookies

Mellanvaran för cookie-parser importeras till filen `app.js`. Det tillåter applikationen att läsa och skriva cookies, som används för att upprätthålla användarsessioner och autentiseringstillstånd. Efter framgångsrik validering av inloggningsuppgifterna väljer applikationen att ställa in säkra, endast HTTP-cookies (`adminSessionId` och `userSessionId`) istället för att upprätthålla sessionstillstånd med sessionsidentifikatorer på serversidan eller förlita sig på klient-side lagringslösningar som `localStorage`. Dessa cookies är konfigurerade för att endast vara HTTP, vilket förbättrar säkerheten genom att förhindra potentiella skadliga skript på klientsidan från att komma åt dem.

Varje inloggningsrutt utför lösenordsverifiering med hjälp av `bcrypt` och tilldelar en nygenererad sessionstoken till en cookie vid framgång. Sessionstoken förstörs nu efter 12 timmar, men denna tidsperiod kan lätt justeras.

Funktionen `generateSecureToken` som visas i figur 9 genererar en säker JSON

Web Token (JWT) som är designad för att autentisera och överföra information om en användare mellan klienten och servern på ett kompakt och säkert sätt.

A screenshot of a code editor showing a JavaScript function to generate a JWT token. The code is as follows:

```
const token = jwt.sign({  
  id: admin.id,  
  role: admin.role,  
  status: admin.status  
}, secretKey, { expiresIn: '12h' });  
return token;
```

Figur 9

Funktionen tar en enda parameter, `admin`, som förväntas vara ett objekt som innehåller åtminstone administratörens unika identifierare (`id`) och deras roll (`roll`). Funktionen använder `process.env.JWT_SECRET` för att hämta den hemliga nyckeln från miljövariablerna. Detta är nyckeln som används för att signera JWT. Att använda miljövariabler för den hemliga nyckeln är en bästa praxis eftersom det håller känslig information borta från kodbasen.

Metoden `jwt.sign()` anropas med nyttolasten och den hemliga nyckeln.

Nyttolast: JWT:s nyttolast inkluderar administratörs-id, status och roll. Dessa data kommer att kodas in i JWT och kan verifieras och avkodas med den hemliga nyckeln.

När servern verifierar en användares autentiseringsuppgifter och genererar denna token, kan alla efterföljande förfrågningar från klienten helt enkelt inkludera token för att komma åt skyddade rutter eller resurser. Servern kommer att validera token med den hemliga nyckeln och bevilja åtkomst baserat på dess giltighet.

#### 4.2.6 Testning

API-koden granskades av utvecklare med lång erfarenhet av backend-utveckling. Hans anmärkningar, allt gällande säkerhetsfrågor, studerades noggrant och koden fixades därefter.

API-koden granskades också av utvecklare med lång erfarenhet av backend-utveckling. Hans anmärkningar, allt gällande säkerhetsfrågor, studerades noggrant och koden fixades därefter.

Alla slutpunkter för REST API testades med hjälp av ThunderClient och Postman regelbundet under hela utvecklingsprocessen. Dessa verktyg användes också för felsökning. Det kontrollerades hur API:et svarar under olika förhållanden, till exempel hur det hanterar autentiseringsfel, hur det validerar indata, hur det hanterar försök att registrera flera konton med ett e-postmeddelande etc. Både ThunderClient och Postman användes också för debuggin: deras felmeddelanden, tillsammans med konsolloggen, visade sig vara till hjälp för att leta efter fel i koden.

## 4.3 Front-end

I det här avsnittet beskrivs konstruktionen av admin-gränssnittet - front-end-delen av applikationsbygget med React, som förbrukar den ovan beskrivna REST API.

### 4.3.1 Routing

I React-applikationen hanteras navigering mellan olika delar av applikationen av react-router-dom-biblioteket, som är en standard för att hantera routing i React-baserade projekt. Kodavsnittet från App.js på figur 10 visar en routerkomponent lindad runt elementet Routes, som fungerar som en behållare för flera ruttkomponenter. Varje ruttkomponent ansvarar för att rendera olika sidor baserat på URL-sökvägen som användaren kommer åt.

```
function App() {  
  return (  
    <Router>  
      <Routes>  
        <Route path="/login" element={<Login />} />  
        <Route path="/" element={<ProtectedRoute><Dashboard /></ProtectedRoute>} />  
        <Route path="/register" element={<ProtectedRoute><RegPage /></ProtectedRoute>} />  
        <Route path="/confirm" element={<ProtectedRoute><ConfirmRegistration /></ProtectedRoute>} />  
        <Route path="/support" element={<ProtectedRoute><SupportPage /></ProtectedRoute>} />  
        <Route path="/admins" element={<ProtectedRoute><Admins /></ProtectedRoute>} />  
        <Route path="/profile" element={<ProtectedRoute><ProfilePage /></ProtectedRoute>} />  
        <Route path="/users" element={<ProtectedRoute><UsersPage /></ProtectedRoute>} />  
        <Route path="/security" element={<ProtectedRoute><SecurityPage /></ProtectedRoute>} />  
        <Route path="/forgot-password" element={<ForgotPassword />} />  
        <Route path="/reset-password" element={<ResetPassword />} />  
      </Routes>  
    </Router>  
  );  
}  
  
export default App;
```

Figur 10

Router är en containerkomponent som omsluter hela routinglogiken. Den lyssnar på ändringar i URL:en och säkerställer att rätt komponent renderas. För denna används BrowserRouter (vilket importeras som Router, Routes och Route (`import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';`) används.

Varje Route-komponent inuti Router definierar en specifik väg och React-komponenten som ska renderas när den sökvägen nås. Till exempel, när man kommer åt /login återges inloggningskomponenten.

Användningen av en ProtectedRoute-komponent för vissa sökvägar säkerställer att dessa rutter endast är tillgängliga för autentiserade användare.

Rutterna är organiserade för att underlätta olika användarinteraktioner som att logga in (/login), registrera (/register), visa en instrumentpanel (/) och komma åt administrativa funktioner (/admins, /users osv). Ytterligare rutter stöder proces-

ser för hantering av användarkonton som lösenordsåterställning (/forgot-password och /reset-password).

#### 4.3.2 Komponentarkitektur

Routing som beskrivs ovan kopplar samman alla komponenter som applikationen innehåller. För närvarande består admin-gränssnittet av elva olika sidor och fyra komponenter som återanvänds på de flesta sidorna. Många sidor innehåller också funktioner som är specifika för dessa sidor och kod för vilken ingår direkt i filen för respektive sida.

Alla programmets filer är organiserade i src-katalogen, med två huvudsakliga underkataloger: sidor och komponenter.

Pages Directory innehåller elva distinkta sidor, som var och en motsvarar en unik rutt i applikationen. Dessa sidor är utformade för att tjäna specifika funktioner, från användarinteraktioner till datapresentation, i linje med applikationens olika behov.

Komponentkatalogen innehåller fyra återanvändbara komponenter som används flitigt på flera sidor. Dessa komponenter ger gemensam funktionalitet och stil som är centrala för applikationens användargränssnitt, vilket säkerställer ett konsekvent utseende och känsla samtidigt som man undviker redundans i koden.

Vissa sidor kräver specialfunktioner som är unika för sina respektive sammanhang. I sådana fall bäddas den specifika koden in direkt i sidans fil.

#### 4.3.3 Komponenter

ProtectedRoute Component är avgörande för att kontrollera åtkomsten till vissa delar av en webbapplikation, vilket säkerställer att endast autentiserade användare kan navigera till skyddade rutter. Koden för denna komponent kommer inte att visas av säkerhetsskäl, men i allmänhet kontrolleras om användare som försöker öppna en skyddad rutt är autentiserad. Om användaren är autentiserad, återger ProtectedRoute-komponenten sina underordnade. Detta möjliggör villkorlig rendering baserat på användarautentiseringsstatus, vilket är avgörande för att upprätthålla säkerheten och integriteten för användarspecifika data i applikationen. Användare som inte är autentiserade omdirigeras till inloggningssidan.

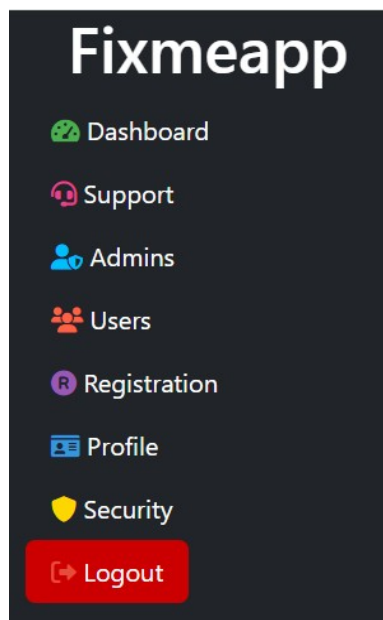
Sidfotskomponenten är den enklaste. Den innehåller lite information om appen och den finns med på alla sidor i applikationen.

NavigationComp är en funktionell komponent som underlättar navigering i applikationen.

Komponenten använder krokar `useLocation` och `useNavigate` från React Router för att hantera routingåtgärder respektive detektera den aktuella ruten, vilket möjliggör villkorlig stil och funktionalitet baserat på användarens navigeringsväg.

UseState-kroken används för att hantera komponentens tillstånd för interaktivitet – specifikt för att ändra knappstilen när en användare håller muspekaren över utloggningssknappen eller över länkarna i navigeringen.

Denna komponent hanterar även utloggning. Figur 11 visar hur navigationsskomponenten ser ut.



Figur 11

Fa-fa-ikoner i olika färger används för länkar i navigeringen. Följande kodavsnitt visar hur det är gjort:

```
<li className="nav-item">
  <Link to="/register" className={`nav-link text-white ${location.pathname === '/register' ? 'active' : ''}`}>
    <i className="fa fa-registered" style={{color: '#9B59B6'}}></i> Registration
  </Link>
</li>
<li className="nav-item">
  <Link to="/profile" className={`nav-link text-white ${location.pathname === '/profile' ? 'active' : ''}`}>
    <i className="fa fa-id-card" style={{color: '#3498DB'}}></i> Profile
  </Link>
</li>
```

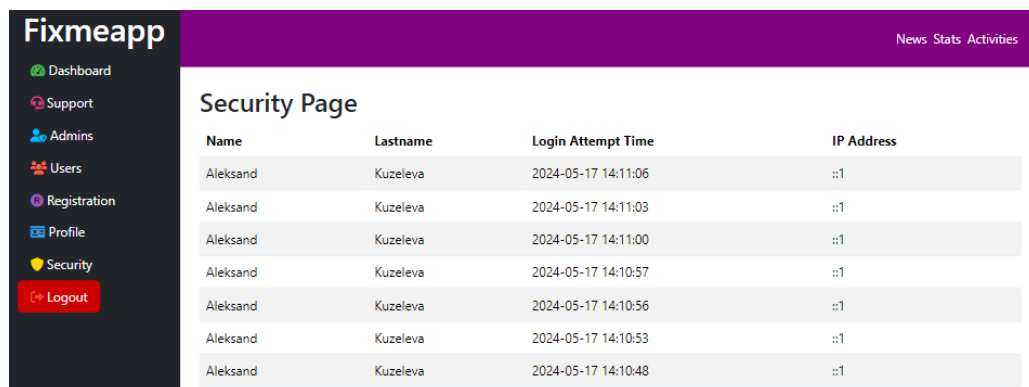
Figur 12

Här importeras ikonerna för registrering och länkar till profilsidan och de olika färgerna för ikoner ställs in, medan texten på alla länkar är vit.

Den sista komponenten – HeaderComp används för att skapa header överst på sidan. Det finns länkar till sidorna nyheter, aktiviteter och statistik. Dessa sidor

finns inte, men det antas att det senare kommer att publiceras nyheter och annan information som är viktig för administratörer av applikationen.

Figur 13 nedan visar den del av säkerhetssidan som visar de senaste inloggningsförsöken, samt huvudnavigering till vänster och header.



The screenshot shows the Fixmeapp interface. On the left is a dark sidebar with navigation links: Dashboard, Support, Admins, Users, Registration, Profile, Security, and a red Logout button. The top header is purple with 'Fixmeapp' on the left and 'News Stats Activities' on the right. The main content area is titled 'Security Page' and contains a table of login attempts.

Name	Lastname	Login Attempt Time	IP Address
Aleksand	Kuzeleva	2024-05-17 14:11:06	::1
Aleksand	Kuzeleva	2024-05-17 14:11:03	::1
Aleksand	Kuzeleva	2024-05-17 14:11:00	::1
Aleksand	Kuzeleva	2024-05-17 14:10:57	::1
Aleksand	Kuzeleva	2024-05-17 14:10:56	::1
Aleksand	Kuzeleva	2024-05-17 14:10:53	::1
Aleksand	Kuzeleva	2024-05-17 14:10:48	::1

Figur 13

IP-adressen ::1 används för loopback-adressen - det vill säga adresser som refererar till själva den lokala maskinen. Lista med ::1 IP-adresser betyder att alla angivna inloggningsförsök gjordes från den lokala maskinen, för teständamål.

#### 4.3.4 Sidor

##### 4.3.4.1 Inloggninssida

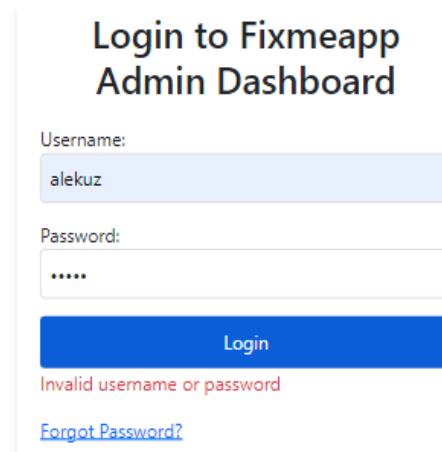
Autentiseringsprocessen på inloggningssidan börjar när användaren skickar in sina referenser (användarnamn och lösenord), hanteras av handleSubmit asynkron funktion. Denna funktion förhindrar standardinlämningshändelsen för att hantera processen programmatiskt. Den gör en POST-begäran till backend-serverns slutpunkt med Axios, och skickar användarnamnet och lösenordet som nyttolast. Svaret från servern förväntas inkludera användarens administrativa status och en identifierare (id), som sedan används för att validera sessionen och hantera användarnavigering.

Felhantering implementeras i handleSubmit-funktionen. Om servern upptäcker ogiltiga autentiseringsuppgifter eller ett inaktivt konto, uppdaterar komponenten feltilståndet för att visa ett relevant meddelande.

Efter framgångsrik autentisering navigeras användningen till huvudinstrumentpanelen med hjälp av useNavigate-kroken från React Router.management.

Om autentiseringen misslyckades ser användaren ett felmeddelande och kan klicka på en länk till en sida där man kan ändra lösenord. Inloggningsformuläret med felmeddelande visas nedan:

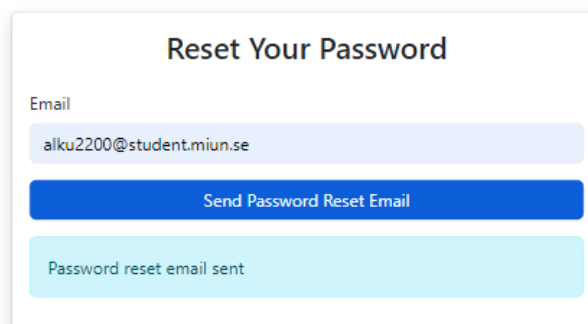




Figur 14

#### 4.3.4.2 Sidan för lösenordsändring

Administratörer som har glömt sina lösenord kan återställa dem med hjälp av detta formulär:



Figur 15

När användaren anger e-postadress och skickar formuläret, skickas POST-begäran med e-postadress i payload:

```
const handleSubmit = async (e) => {  
  e.preventDefault();  
  try {  
    await axios.post('http://localhost:3006/admins/forgot-password', { email });  
    setMessage('Password reset email sent');  
  } catch (error) {  
    setMessage('Error sending password reset email');  
  }  
}
```

Figur 16

När detta händer kontrollerar REST API:t på backend om det mottagna e-postmeddelandet finns i databasen. Om det inte gör det returneras felmeddelandet. Om e-post hittas genereras token som kommer att förfalla efter trettio tillfällen. Det betyder att länken för att återställa lösenordet endast får användas inom en halvtimme. E-post med en länk för återställning av lösenord och en to-

ken ställs sedan in till admin. När användaren klickar på länken öppnas en sida där man kan ange ett nytt lösenord.

När detta händer kontrollerar REST API:t på backend om det mottagna e-postmeddelandet finns i databasen. Om det inte gör det returneras felmeddelandet. Om e-post hittas genereras token som kommer att förfalla efter trettio tillfällen. Det betyder att länken för att återställa lösenordet endast får användas inom en halvtimme. E-post med en länk för återställning av lösenord och en token ställs sedan in till admin. När användaren klickar på länken öppnas en sida där man kan ange ett nytt lösenord. När lösenordet har angetts och administratören klickar på knappen Återställ lösenord, görs ytterligare en POST-begäran som visas i figur 17:

```
try {  
  await axios.post('http://localhost:3006/admins/password-reset', { email, token, newPassword });  
  setMessage('Password reset successful');  
  navigate('/login');  
}
```

Figur 17

Motsvarande funktion, eller begäranhanterare på backend-sidan förväntar sig att förfrågningskroppen ska innehålla e-post, token och newPassword. Om alla tre är närvarande, skickar den en fråga till databasen med hjälp av den angivna e-posten och token för att hitta en motsvarande administratör vars resetPassword-Token inte har löpt ut. Lösenordsåterställning är omöjlig med utgången token.

Om det uppstår ett fel under databasfrågan loggas felet och skickar ett 500-statussvar med ett felmeddelande.

Om ingen administratörsanvändare hittas som matchar de angivna autentiseringsuppgifterna, skickar den ett 400-statussvar som indikerar en ogiltig eller utgången token. Efter framgångsrik validering av token hashas den det nya lösenordet med hjälp av bcrypt för säkerhet.

Den uppdaterar sedan administratörens lösenord i databasen med det nyligen hashade lösenordet. Dessutom rensar den fälten resetPasswordToken och resetPasswordExpires för att indikera att token har använts och inte längre är giltig. Koden för denna begäranhanterare visas i figur 18:

```
router.post('/password-reset', async (req, res) => {
  const { email, token, newPassword } = req.body;

  pool.query('SELECT * FROM admins WHERE email = ? AND resetPasswordToken = ? AND resetPasswordExpires > ?', [email, token, Date.now()], async (error, results) => {
    if (error) {
      console.error(error);
      return res.status(500).json({ message: 'Error processing password reset request' });
    }

    const [admin] = results;

    if (!admin) {
      return res.status(400).json({ message: 'Invalid or expired password reset token' });
    }

    // Hash the new password
    const hashedPassword = await bcrypt.hash(newPassword, 10);

    // Update the password in the database
    pool.query('UPDATE admins SET password = ?, resetPasswordToken = NULL, resetPasswordExpires = NULL WHERE email = ?', [hashedPassword, email], (error, results) => {
      if (error) {
        console.error(error);
        return res.status(500).json({ message: 'Error processing password reset request', error: error.message });
      }

      if (results.affectedRows === 0) {
        return res.status(500).json({ message: 'Error updating admin' });
      }

      res.json({ message: 'Password reset successful' });
    });
  });
});
```

Figur 18

#### 4.3.4.3 Dashboard

På instrumentpanelen finns implementerad funktionalitet för att visa lite statistik: aktuellt antal administratörer, kundsupportpersonal och användare. Detta görs med hjälp av följande funktion:

```
function Dashboard() {
  const [stats, setStats] = useState({
    admins: 0,
    supportAdmins: 0,
    users: 0,
  });

  useEffect(() => {
    const fetchStats = async () => {
      try {
        const { data: admins } = await axios.get('http://localhost:3006/admins');
        const { data: users } = await axios.get('http://localhost:3006/users');
        const adminCount = admins.length;
        const supportAdminCount = admins.filter(admin => admin.role === 'support').length;
        const userCount = users.length;
        setStats({ admins: adminCount, supportAdmins: supportAdminCount, users: userCount });
      } catch (error) {
        console.error('Error fetching stats:', error);
      }
    };

    fetchStats();
  }, []);
}
```

Figur 19

UseState hook initierar en tillståndsvariabel stats med ett objekt som innehåller initiala värden (0) för olika statistik, såsom antalet administratörer, supportadministratörer och användare. setStats används för att uppdatera tillståndsvariabel stats.

Den asynkrona funktionen fetchStats är definierad för att hämta statistikdata från servern. Den här funktionen använder axios för att göra asynkrona HTTP-förfrågningar till slutpunkter (/admins och /users) på den lokala värdservern. Efter att ha fått svar beräknar den statistik som antalet administratörer, support-

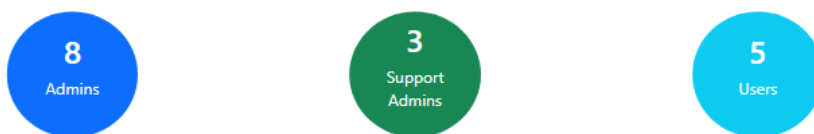
administratörer och användare. Slutligen uppdaterar den statistiktillståndet med den nya statistiken med `setStats`.

Dashboard-sidan, såväl som hela applikationens front-end, är stil med hjälp av ramverket Bootstrap. En del av koden där Bootstrap används visas nedan:

```
<h2>Currently there are:</h2>
<div className="d-flex justify-content-around">
  <div className="text-center">
    <div className="circle bg-primary text-white p-5 d-flex align-items-center justify-content-center flex-column" style={{ borderRadius: '50%', width: '130px', height: '130px', margin: '10px 0' }}>
      <h2>{stats.admins}</h2>
      <p>Admins</p>
    </div>
  </div>
  <div className="text-center">
    <div className="circle bg-success text-white p-5 d-flex align-items-center justify-content-center flex-column" style={{ borderRadius: '50%', width: '130px', height: '130px', margin: '10px 0' }}>
      <h2>{stats.supportAdmins}</h2>
      <p>Support Admins</p>
    </div>
  </div>
  <div className="text-center">
    <div className="circle bg-info text-white p-5 d-flex align-items-center justify-content-center flex-column" style={{ borderRadius: '50%', width: '130px', height: '130px', margin: '10px 0' }}>
      <h2>{stats.users}</h2>
      <p>Users</p>
    </div>
  </div>
</div>
```

Figur 20

Koden på figur 20 gör det möjligt att visa statistik på följande sätt:



Figur 21

#### 4.3.4.4 Sidan admins

På administratörssidan finns en lista över alla admins (de som har status admins) med sina data, som är redigerbara.

Genom asynkrona funktioner kommunicerar sidan med en serverslutpunkt för att hämta och uppdatera admindata. Initial data hämtas vid komponentmontering (`useEffect`), och ändringar av admindata hanteras genom specifika funktioner som uppdaterar data på serversidan via POST- och PUT-förfrågningar.

Det är möjligt att redigera alla administratörsdata, inklusive bilder. Viktig funktion är att ändra status för en administratör: att göra den aktiv, inaktiv eller avstängd. Varje status har olika färgkoder.

Det är inte möjligt att ta bort admins i admin-gränssnittet. När projektet diskuterades i detalj beslutades det att inga administratörer eller användare kommer att tas bort från databasen, även de som har varit avstängda under lång tid. Men i REST API finns det funktionalitet som gör det möjligt att permanent ta bort administratörer.

På sidor presenteras användare och supportdata från klientsupportadministratörer och användare på exakt samma sätt som på adminsidan, med samma funktionalitet för redigering.

#### **4.3.5 Testning**

Följande tester gjordes för att säkerställa funktionalitet och överensstämmelse med webbstandarder.

Funktionstestning. Nyckelfunktioner, särskilt formuläroperationer som datainlämning och bilduppladdning, testades omfattande för att säkerställa att de beter sig som förväntat.

W3C-validering. Applikationen testades med W3C Validator för att säkerställa att den följer de senaste webbstandarderna. Validator visade inga fel på alla sidor i appen.

## 5 Resultat

Ett av de primära målen med projektet var att utveckla en databas som effektivt lagrar och hanterar data som handlar om både kunder och administratörer. Databasen implementerades framgångsrikt, med kapacitet att logga användarinloggningstider och registrera misslyckade inloggningsförsök, vilket förbättrade säkerhetsåtgärderna avsevärt.

Under projektetarbete utökades databasens omfattning avsevärt utöver dess ursprungliga design. Den omfattar nu ett brett utbud av funktioner inklusive hantering av tjänster, bokningar, prisinformation, betyg och kundlojalitetsdata.

Utvecklingen av databasen var en pågående process under hela projektets gång. Databasen designades om flera gånger för att bättre anpassas till projektets föränderliga behov. Det tjänar nu sitt syfte väl, men det är troligt att det kommer att genomgå ytterligare modifieringar när nya behov och utmaningar uppstår.

Ett RESTful API utvecklades för att hantera användare och administratörer, inklusive autentisering och inloggning. Detta API möjliggör viktiga operationer som att skapa, modifiera och ta bort användare, vilket effektivt uppfyller kraven på administrativa åtgärder.

Registreringsprocesser för användare och administratörer implementerades framgångsrikt. Registrering kräver e-postbekräftelse, vilket förbättrar säkerheten och integriteten för användardatahantering. REST API hanterar även bilduppladdningar, lösenordsändringar och registrering av misslyckade inloggningsförsök.

Ett administratörsgränssnitt skapades med hjälp av React. Detta gränssnitt möjliggör registrering av nya administratörer, visning och redigering av data för administratörer och användare, och övervakning av misslyckade inloggningsförsök.

Administratörsgränssnittet innehåller separata avsnitt för administratörer och supportarbete, vilket säkerställer att datahanteringen är både specifik och säker. Vid senare utvecklingsstadier kommer rollbaserad åtkomst till administratörsgränssnittet att skapas - till exempel kommer kundsupportarbete bara att kunna hantera användarkonton, men inte administratörskonton, etc.

Gränssnittet är endast tillgängligt för autentiserade användare, med säkerhetsåtgärder på plats för att förhindra obehörig åtkomst.

Inloggningssidan innehåller en "Glömt lösenord"-länk, som leder till en sida för lösenordsåterställning.

Autentiserade användare kan ändra sina lösenord direkt från sina profilsidor, en funktion som förbättrar användarnas autonomi och säkerhet.

Applikationen använder formulär för datainmatning och modifiering, som testades och fungerar som förväntat.

## 6 Slutsatser

Sammanfattningsvis kan det konstateras att projektet framgångsrikt uppnådde sina initiala mål: en fullständig implementation av databasen, skapandet av ett REST API och utvecklingen av ett administrativt gränssnitt. Att skapa databasen och utveckla applikationen från grunden var en betydande utmaning, särskilt med tanke på att jag leddes av en mentor med begränsad utvecklingserfarenhet. Dessa utmaningar mildrades dock avsevärt under projektets gång genom tillägget av en erfaren utvecklare till teamet. Hans insikter, särskilt vad gäller säkerhetsförbättringar, var ovärderliga och bidrog avsevärt till projektets framgång. Dessutom förmedlade han avgörande kunskap och metoder som berikade hela utvecklingsupplevelsen.

Trots att projektet nu befinner sig i ett funktionellt och stabilt tillstånd finns det fortfarande en betydande potential för förbättringar och expansion. Framtida utvecklingsinsatser kommer att fokusera på att finslipa befintliga funktioner, förbättra användarupplevelsen och kontinuerligt integrera avancerade säkerhetsåtgärder för att hålla jämna steg med den ständigt föränderliga hotmiljön.

Valet av Express.js och React för projektet visade sig vara korrekt och väl anpassat till våra behov. Express.js erbjöd en smidig och effektiv hantering av backend-funktionalitet, medan React möjliggjorde utvecklingen av en responsiv och användarvänlig frontend. Genom att arbeta med dessa teknologier har jag lärt mig mycket om modern webbutveckling och fördjupat min förståelse för hur man bygger robusta och skalbara applikationer. Detta har varit en ovärderlig erfarenhet som har berikat min professionella utveckling och ökat min kompetens inom området.

### 6.1 Etisk och social diskussion

I etik- och social diskussionsavsnittet framträder en mångfacetterad bild av de etiska övervägandena som präglade utvecklingen av applikationen. Detta återspeglar ett djupt engagemang för inkludering och integritet. Ett betydande beslut som togs var att inte lagra användarnas könsinformation i databasen. Genom att göra detta val betonades vikten av att erkänna och främja en bredare samhällelig förändring som innebär att attribut som hårtyp inte automatiskt kopplas till kön. Detta främjar ett mer inkluderande tillvägagångssätt som undviker att påtvinga binära könsnormer.



Utöver detta strävar Fixmeapp också efter att skapa ekonomiska möjligheter för marginaliserade grupper, exempelvis flyktingar som är frisörer och som söker en stabil inkomst. Genom att erbjuda en plattform där de kan få kontakt med potentiella kunder spelar appen en avgörande roll för att integrera dessa individer i samhället och hjälpa dem att uppnå ekonomiskt oberoende.

När det kommer till efterlevnad av dataskyddslagar som GDPR, harmoniserar beslutet att inte lagra information om kön i databasen inte bara med etiska överväganden om integritet och inkludering utan minimerar också risken för potentiella känsliga dataintrång. Framåt kommer fortsatta ansträngningar för att förbättra dataskyddet att vara av avgörande betydelse, särskilt eftersom applikationen fortsätter att skalas och hantera ökande mängder användardata.

## Källförteckning

- [1] Amazon Web Services. "Structured query language (SQL)" [https://aws.amazon.com/what-is/sql/#:~:text=Structured%20query%20language%20\(SQL\)%20is%20a%20standard%20language%20for%20database,undergoes%20continual%20upgrades%20and%20improvements](https://aws.amazon.com/what-is/sql/#:~:text=Structured%20query%20language%20(SQL)%20is%20a%20standard%20language%20for%20database,undergoes%20continual%20upgrades%20and%20improvements). Hämtad 2024-05-13
- [2] Amazon Web Services. What is an API?" <https://aws.amazon.com/what-is/api/> Hämtad 2024-05-13
- [3] Fortinet. "What is an API Key?" <https://www.fortinet.com/resources/cyberglossary/api-key> Hämtad 2024-05-13
- [4] Geeks for Geeks, "Axios in React: A Guide for Beginners" <https://www.geeksforgeeks.org/axios-in-react-a-guide-for-beginners/> Hämtad 2024-05-13
- [5] Hostinger. "What is Bootstrap - Everything you need to know." <https://www.hostinger.com/tutorials/what-is-bootstrap/> Hämtad 2024-05-13
- [6] Kinsta. "What is Node.js and why you should use it." <https://kinsta.com/knowledgebase/what-is-node-js/> Hämtad 2024-05-13
- [7] Kinsta. "What is React.js? A look at the popular JavaScript library." <https://kinsta.com/knowledgebase/what-is-react-js/> Hämtad 2024-05-13
- [8] Mozilla Developer Network. "Express/Node introduction" [https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express\\_Nodejs/Introduction](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction) Hämtad 2024-05-13
- [9] Nanonets. "Understanding SendGrid: A Guide to Email Automation" <https://nanonets.com/blog/what-is-sendgrid-email-automation/> Hämtad 2024-05-13
- [10] npm. "bcrypt." <https://www.npmjs.com/package/bcrypt> Hämtad 2024-05-13
- [11] Our Coding Club. "Intro to Github for version control" <https://ourcodingclub.github.io/tutorials/git/> Hämtad 2024-05-13

- [12] Refine Development Inc. "4 Ways to Generate UUIDs in Node.js"  
<https://refine.dev/blog/node-js-uuid/#why-use-uuids-in-your-nodejs-projects> Hämtad 2024-05-13