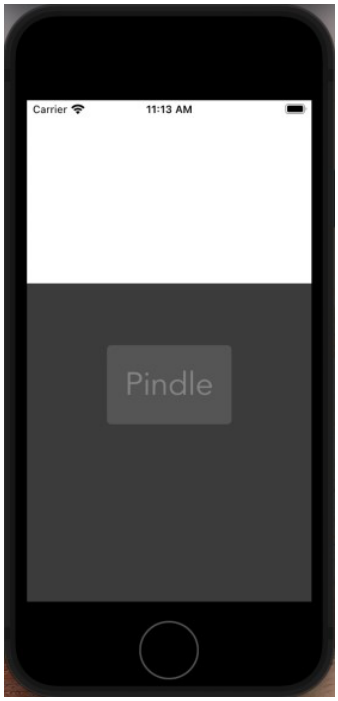


Implementation och design av en databas och ett REST API byggd i .NET Core

Examensarbete inom högskoleingenjörsprogrammet datateknik

Cynthia Kozma
Petra Nisan

INSTITUTIONEN FÖR DATA- OCH INFORMATIONSTEKNIK

CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2022 www.chalmers.se

Examensarbete 2022

Implementation och design av en databas och ett REST API byggd i .NET Core

Cynthia Kozma
Petra Nisan

Institutionen för Data- och Informationsteknik
Chalmers Tekniska Högskola
Göteborg, Sverige 2022
www.chalmers.se

Implementation och design av en databas och ett REST API byggd i .NET Core

Handledare:
Morten Fjeld, Professor, avdelningen för Interaktionsdesign och Software Engineering, Institutionen för data- och informationsteknik. Chalmers Tekniska Högskolan.

Examinator:
Jonas Duregård, Universitetslektor på avdelningen för Computer Science, Institutionen för data- och informationsteknik. Chalmers Tekniska Högskolan.

Institutionen för data- och informationsteknik
Chalmers Tekniska Högskolan
SE-412 96 Göteborg
Telefon: +46 (0)31-772 1000

Omslag: Alla figurer i denna rapport är designade av författarna.

Institutionen för data- och informationsteknik
Göteborg, Sverige 2022

Abstract

Almost immediately after computers became common in the 1960s, the need for secure data storage developed. Gradually, different types of databases were developed and today the relational database is the most common type. For today's websites and applications, an advanced database is required to fulfill all the requirements. This report describes how a searchable database has been developed for a prototype mobile-application called Pindle. Pindle is an ongoing project at the company pinDeliver that has a collaboration with the company Elicit Software. The purpose of this project was to investigate how storage of routes can be made efficient and searchable in a database. Additionally, it was determined which data types were most suitable to use in this database that would fulfill the purpose of the application. It was also investigated whether it would be possible to use only two tables in the database and still serve the purpose. Finally, it was investigated whether it would be possible to implement an automatic message sent in a group chat.

The project is about creating a suitable database design for the multiple functions that will be available in the application such as save contacts, retrieve the user's own routes, and optimize them. With the help of a REST API in .NET CORE that creates communication between server and client, various endpoints have been implemented. The code was written in programming languages C# and XAML using the source-code editor Visual Studio Community 2022. The work of the project group was divided into a one-week period based on the agile management methodology. During the project, the couple used Pair Programming as they are quite comfortable with this technology. As a result of this project, a searchable database is created using only two tables that fulfill the purpose of this application-prototype. In addition, an automatic message sent in a group chat was also implemented based on the SignalR framework.

Keywords: Database design, REST API, .NET Core, application, tables, endpoints.

Sammanfattning

När datorer började användas på 1960-talet, uppstod behovet av att spara datan på ett säkert sätt. Så småningom utvecklades olika typer av databaser, i nuläget är relationsdatabas den dominerande typen. De flesta hemsidor och applikationer idag har en avancerad databas för att täcka alla krav. Denna rapport kommer att beskriva hur en sökbar databas har utvecklats för en mobilapplikation-prototyp som kallas för Pindle. Pindle är ett pågående projekt hos företaget pinDeliver som i sin tur har ett samarbete med företaget Elicit Software. Syftet med detta projekt var att undersöka hur lagring av rutter kan göras effektivt och sökbart i en databas. Vidare, undersöktes vilka datatyper som var lämpligast att använda i just denna databas som skulle uppfylla applikationens syfte. Dessutom undersöktes det om det skulle vara möjligt att endast använda två tabeller i databasen men ändå uppfylla syftet. Slutligen, undersöktes om skulle vara möjlig att implementera ett automatisk meddelande som skickas i en gruppchatt.

Projektet går ut på att skapa en lämplig databasdesign för de olika funktioner som applikationen kommer bland annat att ha; spara och hämta kontakter, hämta användarens egna rutter samt optimera dem. För att dessa funktionaliteter ska implementeras behövdes först en kommunikation mellan server och klienter. Denna kommunikation skapades med hjälp av ett RESTful API i .NET Core. Koden för projektet skrevs i programutvecklingsmiljö Visual Studio Community i programmeringsspråken C# och XAML. Projektgruppen arbetade utifrån det agila systemutvecklingsmetoder där arbetet delades upp i veckolånga sprintar. Paret har även använt sig av parprogrammering under hela arbetet eftersom paret är ganska bekväma med denna teknik. Resultat framställs i en sökbar databas skapad med två tabeller som uppfyller applikationens syfte. Dessutom implementerades ett automatisk meddelande som skickas i en gruppchatt utifrån ramverket SignalR.

Nyckelord: Databasdesign, REST API, .NET Core, applikation, tabeller, endpoints.

Förord

Denna rapport är ett kandidatarbete i Datateknik-högskoleingenjörsprogrammet i Chalmers Tekniska Universitet. Detta arbete är skriven av Cynthia Kozma och Petra Nisan under vårtermin 2022.

Först vill vi tacka företagen pinDeliver och Elicit Software för att vi fick arbeta med ett av deras framtida projekt. Vidare vill vi särskilt tacka våra handledare, Jenny Forsberg från pinDeliver för den stöden vi fick under hela arbetet och Morten Fjeld på Chalmers för snabba svar och tillgänglighet vid behov. Vi vill också rikta ett stort tack till studenterna Negin Bakhtiarirad och Mojtaba Ataie i grupp 22 för ett jättelyckat samarbete med chattsystemet.

Vi har även erhållit mycket stöd av familjer och vänner vilket har varit enormt viktigt för slutförandet av detta arbete. Dessa personer har lyst våra dagar när vi har haft det svårt och stöttat oss.

Ordlista:

Affärslogik: de anpassade regler eller algoritmer som hanterar utbytet av information mellan en databas och användargränssnitt

Applikation: Datorprogrampaket som utför en specifik funktion direkt för en slutanvändare eller, i vissa fall, för en annan applikation.

Backend: Ett underliggande system som hanterar allt som händer bakom kulisserna i en applikation såsom en server.

CRUD: En akronym som syftar på fyra funktioner som anses vara nödvändiga för att implementera en beständig lagringsapplikation: skapa, läsa, uppdatera och ta bort.

DTO: Ett objekt som tar hand om transporten av data mellan backend och frontend

Endpoints: Ena änden av en kommunikationskanal. När ett API interagerar med ett annat system anses kontaktpunkterna för denna kommunikation vara endpoints (slutpunkter).

Frontend: Ett grafiskt användargränssnitt till applikationen som användaren kan interagera med.

IP-adress: En uppsättning av nummer som identifierar en anslutning mot internet. En IP-adress är en unik adress som identifierar en enhet på internet eller på ett lokalt nätverk.

Klient - Server: Ett designmönster för att flera programvarukomponenter skall kunna kommunicera med varandra via ett gränssnitt. Oftast är det en server som kommunicerar med flera klienter.

Null: Används i datorprogrammering för ett oinitierat, odefinierad, tomt eller meningslöst värde.

REST API: Står för “Representational State Transfer Application Programming Interface”. Ett applikationsprogrammeringsgränssnitt som tillhandahåller kommunikationen mellan applikationer och databaser.

SQLite: Ett databashanteringssystem som implementerar en fristående och serverlös inbäddad SQLdatabasmotor baserat på SQL.

Xamarin.Forms: Ett gränssnittsverktyg som användes i frontend som gör det möjligt för en utvecklare att skapa utseendet för användargränssnittet som kan delas mellan Android och iOS.

.NET Core: Ett ramverk i form av ett multiplattform i syfte att utveckla applikationer som skall kunna köras på olika operativsystem.

Innehållsförteckning

1. Inledning
Rapporten inleds med en introduktion om företagets bakgrund, syfte, mål samt frågeställningen för projektet och slutligen avgränsningar.

1.1 Bakgrund
Detta projekt är ett samarbete med ett IT företag som heter Elicit Software. Företaget grundades 1999 med målet att utveckla hållbara produkter och tjänster som skapar nytta till både sina kunder och miljön. Elicit har varit med och byggt flera projekt, bland dessa är en webbplats för social utveckling, Antura Projects och Easy-Laser AB. Företaget lanserade även sedan många år tillbaka en välutvecklad leveransplattform, pinDeliver. Plattformen är en molnbaserad leveransplattform som effektiviserar och digitaliserar hela leveransprocessen. Syftet med plattformen är att låta användarna följa transportprocessen hela vägen från plockning till utlämning [1]. Nu vill företaget vidareutveckla pinDeliver genom att erbjuda privatpersoner samma möjlighet att samordna sina resor. Detta är tänkt att utvecklas i en mobilapplikation-prototyp som kallas för Pindle. Prototypen byggs genom en klient-server lösning som gör det möjligt för användare att kommunicera med varandra, spara kontakter och rutter och viktigast av allt optimera de sparade rutterna.

Detta projekt bygger vidare på ett tidigare arbete som gjordes i projektkursen DAT067 tillsammans med andra studenter. I projektkursen implementerades gränssnittet för Pindle samt några funktionaliteter som till exempel lägga till en kontakt, skapa en rutt och optimera rutten. Företaget erbjöd två examensarbeten där grupp 22 tog det ena arbetet som handlar om att skapa ett chattsystem för prototypen Pindle, och vi tog det andra arbetet som handlar om att skapa databasen för Pindle. Databasen för Pindle är skapad med hjälp av SQLite som är ett programvarubibliotek i programspråket C som implementerar en filbaserad SQL-databas [2]. Företaget efterfrågade en så enkel databas som möjligt som ska alltså innehålla så få tabeller som möjligt. Därför ska projektgruppen försöka uppfylla applikationens syfte genom att endast skapa de nödvändiga tabellerna.

1.2 Syfte
Syftet med projektet är att undersöka hur lagring av rutter kan göras effektivt och sökbart i en databas. För att applikationen Pindle ska fungera krävs det att någonting som kallas rutter sparas i databasen. En rutt är ett antal geografiska punkter placerade i en viss ordning på en karta. Dessutom skall en undersökning på lämpliga datatyper för lagring och sökning av positioner och rutter utföras. Utöver databasen och all lagring, kommer ett automatiskt meddelande att implementeras i samarbete med exjobbsteamet 22 för att göra applikationen användarvänlig.

1.3 Frågeställning
Företaget Elicit har ett stort fokus på att göra deras tjänster effektiva samt användarvänliga. Därför var en effektiv och sökbar databas för applikationen Pindle otvivelaktig. För att detta skall uppfyllas krävs det att ett lämpligt ramverk används men även att endast relevant data sparas i databasen.

Med utgångspunkt i detta är frågeställningarna med underfrågeställning följande:

- Kan en sökbar databas i SQLite som integrerar med ett REST API i .NET Core utvecklas till mobilapplikation-prototypen Pindle?
    - Skulle skapandet av endast två tabeller vara tillräcklig för en sådan databas?

- Kan ett automatiskt meddelande som ska skickas i en gruppchatt och är anpassad efter prototypens chattsystem implementeras?

1.4 Mål

Målet med projektet är att skapa och implementera en konkret databasdesign som uppfyller applikationens syfte. Databasen kommer att skapas till en redan befintlig prototyp för att kunna lagra viktiga data från applikationen. Utmaningen i detta projekt är att skapa en databas anpassad efter prototypen och dess funktionaliteter men även att skapa en stabil kommunikation mellan backend och frontend. Målet är även att göra prototypen användarvänlig genom bland annat ett meddelande som ska skickas automatiskt i en gruppchatt för att meddela resten av användarna i chatten att en användare har blivit upphämtad. Databasen skall sedan publiceras i Elicit Softwares servermiljö.

Målen för detta projekt är följande:

- Utforma en lämplig databasdesign för att göra en enkel sökbar databas
- Skapa en effektiv och sökbar databas i SQLite som integrerar med ett REST API i .NET Core
- Skapa ett REST API och lämpliga endpoints som möjliggör kommunikationen samt förfrågningar av data mellan databasen och applikationen
- Implementera ett automatiskt meddelande i realtid utifrån ett lämpligt ramverk som ska skickas i en gruppchatt som tillhör en rutt när en användare skall bli upphämtad

1.5 Avgränsningar

- En redan befintlig prototyp utvecklad i Xamarin.Forms kommer att användas för detta projekt.
- Fokuset kommer inte ligga på att göra databasen säker, alltså kommer alla som har rätt länk till databasen att kunna se innehållet.

2. Teknisk bakgrund

Detta avsnitt går igenom mjukvaran som användes i projektet och hur den är relaterad till arbetets syfte.

2.1 Programmeringsspråket C#

C# är ett utav de populäraste programmeringsspråken. Det är ett objektorienterat språk som främst används i objektorienterade projekt. Eftersom C# är en efterträdare till programmeringsspråket C++ samt besläktad med Java, är språket hyfsat enkelt att lära sig. Av den anledningen är C# ett populärt språk som idag har en hög efterfråga.

Fördelar med C# som upptäcktes under projektet:

- Enkelt språk att programmera till följd av enkel struktur på koden.
- Ett brett utbud av kodbibliotek som öppen källkod, såsom NuGet-paket.
- Många färdiga och nödvändiga metoder.

Nackdelar med C# som upptäcktes under projektet:

- Koden behöver kompileras efter varje liten ändring [3].

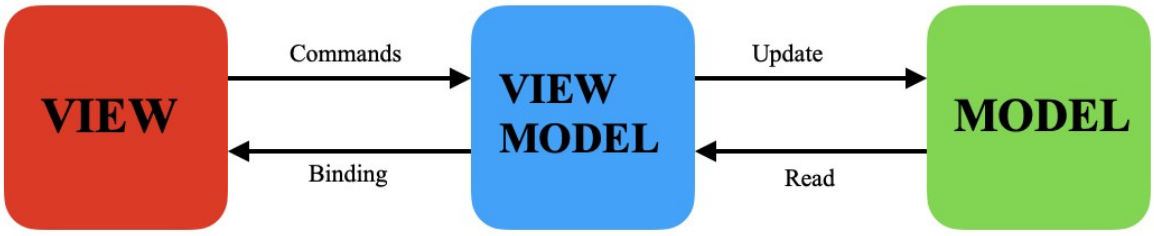
2.2 MVVM - Ett mjukvarudesign mönster

MVVM (model-view-viewmodel), är ett mjukvaru-arkitektoniskt mönster som underlättar separationen av utvecklingen av det grafiska användargränssnittet enligt [4]. Detta innebär att en vy inte är beroende av backend-logiken eller någon specifik modell. MVVM-modeller bör öka programkodens läsbarhet och förbättra organisationen av filerna. Designmönstret är uppdelat i följande 3 delar, modell, vy och vymodell (se figur 2.1).

Modell-klasser är icke-visuella klasser som kapslar in applikationens data och representerar applikationens bedömningsmodell. Modell-klasser innehåller inte någon affärslogik och bör alltså vara så "ren" som möjligt enligt [4]. Modellen har därför inga referenser till vyn eller vymodeller samt inga kunskaper om hur den används.

Vymodell-klasser definieras enligt [4] som en abstrakt del av hanteringen av modellerna och deras egenskaper som visas på användarens skärm. Dessa klasser fungerar som en mellanhand mellan vyn och modeller, då den hämtar data från modellen och skickar den vidare till vyn. Vymodellen ansvarar för att meddela vyn om det har skett ändringar i utseendet genom bindings.

Vy-klasser är gränssnittet som användaren ser och är därför ansvarig för att visa data från modellen och vymodellen. Vyn ska innehålla så lite bakomliggande kod och så lite affärslogik som möjligt enligt [4]. En vy kan antingen ha en egen vymodell eller ärva den från en annan vymodell. Utseendet för vyn ändras av vymodellen med hjälp av metoder och bindings som meddelar vyn om en specifik ändring.



Figur 2.1, illustrerar MVVM mönstret och hur de olika klasserna hänger ihop

2.3 Databas

En databas är en organiserad samling som lagrar data, oftast elektroniskt i ett datorsystem. Databasen styrs sedan med hjälp av ett databashanteringssystem (DBMS). All data, DBMS och de applikationer som är associerade med DBMS är tillsammans definitionen av ett databassystem, som även kallas en databas [5]. En databas innefattar rader (posts) och kolumner i tabeller som hanterar lagringen av data. Denna data kan då lättare kommas åt och modifieras.

Databaser verifierar att korrekt data som skall lagras passar in i databasens struktur, dvs i tabellerna. För att databasen skall kunna säkerställa dessa aspekter krävs det att varje ändring eller förfrågan överstämmer med en uppsättning av regler som kallas för ACID, "Atomicity, Consistency, Isolation, Durability" [6]. Dessa regler är följande:

- **Atomicitet**: om en liten ändring av data i databasen misslyckas kommer hela ändringen att misslyckas. Informationen kommer därmed att förbli som den var innan den misslyckade ändringen. Detta för att hindra ofullständiga data.

- **Konsistens**: innan information i databasen kan uppdateras måste all data kontrolleras av en uppsättning av regler.

- **Isolering**: en databas kan tillåta flera ändringar parallellt, dock är varje ändring isolerad från de andra.

- **Hållbarhet**: när en ändring har gått igenom är all data säker, även om det skulle uppkomma fel i systemet

2.3.1 Databashanterare

En databashanterare är en programvara som hanterar strukturen hos en databas som lagrar data. Den sköter alla primära aspekter av en databas, dvs upprätthåller dataintegritet, skapa tabeller samt relationer etc. Funktionaliteter som att uppdatera, lagra och söka i databasen är några av de huvudsakliga funktionerna för en databashanterare. Det är även ett enkelt sätt att kontinuerligt ha kontroll över att databasen är uppdaterad och innehåller korrekt information. De flesta databashanterare har dessutom ett gränssnitt som gör det enklare för utvecklaren att se databasen och dess lagrad data visuellt [7].

- **DBeaver**: en databashanterare som användes på macOS under projektets gång. Den stödjer alla populära databaser såsom MySQL, PostgreSQL, Oracle, SQLite etc. DBeaver funktioner inkluderar bland annat utförande av SQL-frågor och hantering av databasstrukturer. Detta innefattar hantering av tabeller och kolumner som främst användes.

- **SQL Server Management Studio (SSMS)**: en databashanterare i syfte att hantera SQLinfrastrukturer, likt DBeaver. SSMS är Microsofts egna databashanterare och används därför i detta projekt för Windows.

2.3.2 SQLite

SQLite är ett databashanteringssystem som implementerar en fristående och serverlös inbäddad SQLdatabasmotor baserat på SQL. Det innebär att SQLite skriver och läser direkt till diskfiler och inte via en separat server. En diskfil innehåller en komplett SQL-databas med flera tabeller, vyer osv som ligger lokalt på datorn. SQLite testar källkoden noggrant med hjälp av bland annat ACID reglerna för att allt ska vara säkert och pålitligt [8].

Fördelar med SQLite:

- Stödjer C#
- Enkel att distribuera
- Databasen blir en integrerad del av projektet och därmed eliminerar resurskrävande fristående processer

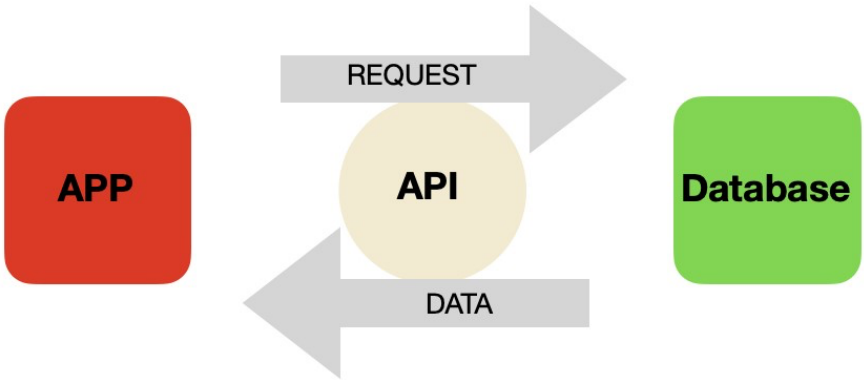
2.4 REST API

API som står för "Application Programming Interface" är ett applikationsprogrammeringsgränssnitt som används för att program, system och applikationer enkelt och snabbt ska kunna kommunicera med varandra [9]. Med andra ord agerar API som en budbärare som hanterar förfrågningar mellan olika system (se figur 2.2). Dessa förfrågningar implementeras som endpoints där varje endpoint har en funktion. Exempelvis finns det en endpoint för att fråga databasen om data från applikationen och en annan endpoint som sparar data från applikationen till databasen.

API Integration ger möjligheten till att:

- skicka och hämta information
- ansluta sig till molnet där all information sparas

Ett REST API, där REST står för "REpresentational State Transfer", är ett sätt att bygga ett API som baseras på HTTP-funktioner. Fördelen med REST är att den utnyttjar så mycket av de HTTP-funktioner som möjligt istället för att hitta nya sätt att hantera dataöverföringen på. En annan fördel är att den använder en standard struktur för webben som är populär bland utvecklare och programmerare, vilket gör API enkelt att förstå. Denna fördel kan även anses vara en nackdel eftersom det blir svårt i vissa situationer att hitta ultimata lösningar för all typ av kommunikation som använder en så pass enkel struktur som REST API gör.



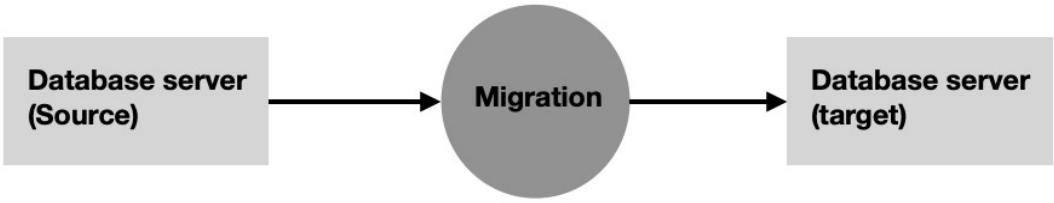
Figur 2.2, en illustration över hur en API förfrågan går till. Den röda fyrkanten är applikationen som skickar en förfrågan till databasen, den gröna fyrkanten, via ett API anrop, Databasen svarar på förfrågan och skickar tillbaka data som efterfrågades.

2.5 Data Transfer Objects (DTO)

Data transfer objects, som på svenska är dataöverföringsobjekt, är ett objekt som tar hand om transporten av data mellan olika system. Den här tekniken användes för att underlätta kommunikationen mellan servern (databasen) och API utan att potentiellt överge känsliga data [10]. Med hjälp av DTO-objekt överförs endast nödvändiga data. Exempelvis när en ny kontakt ska skapas skall man ej tilldela ett id värde utan det får kontakten automatiskt när den sparas i databasen. Då användes ett DTO-objekt som innehåller alla kolumner som finns i kontakt tabellen förutom id. DTO-objekt hanterar inte någon logik utan används främst vid lagring, hämtning, serialisering och deserialisering av data.

2.6 Migrationer

Databasmigrering är en process för att migrera data från en eller flera databaser till en eller flera destinationer (se figur 2.3). Migrationer är användbara eftersom de underlättar för utvecklaren att göra ändringar i databasen på ett säkert sätt [11]. Dessa migrationer skapades i Visual Studio terminalen och automatiskt skapas en fil med migrationen. När en ändring gjordes i en av tabellerna som borde skickas till databasen behövdes en ny migration skapas med den nya ändringen.



Figur 2.3, en illustration av hur migrationer används. Migrationer som innehåller data, skickas från databasen till applikationen när en API förfråga sker

Database server (source): Databasen som innehåller all data som ska migreras till en eller flera destinationer. I detta fall är det databasen i SQLite.

Database server (target): Destinationen som tar emot all data som migrerats från en eller flera databaser. I detta fall är det applikationen Pindle som fick all data från databasen.

2.7 .NET Core

.NET Core är ett ramverk med öppen källkod utvecklat av Microsoft i syfte att utveckla bland annat webbsidor och mobilapplikationer. Ramverket är en så kallad multiplattform vilket innebär att den kan köras på olika operativsystem. Datorprogram som är avsedda att vara multiplattform kräver att de körs i en utvecklingsmiljö som är avsedd att stödja multiplattform, med denna avsikt användes Visual Studio [12].

Funktioner som .NET Core tillhandahåller är bland annat NuGet-paket vilket är en pakethanterare för .NET som en utvecklare kan skapa, dela och konsumera användbar kod. Paketet innehåller en enda zip fil med kompilerad kod och andra filer som paketet behöver. Utvecklaren lägger till dessa NuGet-paket i sitt projekt och kan sedan anropa ett pakets funktionalitet i sin projektkod. NuGet själv hanterar sedan alla mellanliggande detaljer. Huvudsakliga syftet med NuGet-paket är att de konfigurerar automatiskt

projektet genom att lägga till referenser till nödvändiga sammansättningar, skapa och lägga till projektfiler etc [13].

Fördelar med .NET Core:

- Bra verktyg vid användandet av ett REST API
- Objektorienterat ramverk
- Innehåller verktyg som förenklar modern webbutveckling

2.8 JSON

JSON som står för JavaScript Object Notation, är ett lättviktsformat för datautbyte. Den är både lätt för människor att läsa och för maskiner att generera. Den är skriven i programmeringsspråket JavaScript och är språkoberoende [14]. JSON-dokument är en informationsresurs och består därmed av fält som består av nyckel-värdeobjekt. Fältnamnet fungerar som en nyckel för fältet och måste därför vara unikt [15]. För att kunna bearbeta JSON i ett projekt används oftast namnområdet "namespace", *System.Text.Json* som ger högpresterande, låg-allokerande och standard-kompatibla funktioner (se figur 2.4). Den inkluderar även serialisering av objekt till JSON-text och deserialisering av JSON-text till objekt [16].

```
using System.Text.Json;
```

Figur 2.4, en illustration över namnområdet System.Text.Json som användes i projektet

2.8.1 Serialisering & deserialisering

Serialisering är processen att konvertera dataobjekt (en kombination av kod och data representerade inom ett område av datalagring) till en serie av bytes som sparar objektets tillstånd i en lätt överförbar form. Med andra ord, används serialisering för att skriva om JSON till en sträng i form av ett JSON-objekt. I detta projekt valde gruppen att serialisera ett objekt genom att anropa metoden *JsonConvert.SerializeObject* och skicka in objektet som ska serialiseras som en parameter (se figur 2.5).

```
var strInfo = JsonConvert.SerializeObject(contact);
```

Figur 2.5, en kodrad från projektet där ett objekt serialiseras

Deserialisering är den motsatta processen till serialisering där en sträng skrivs om i form av ett JSON-dokument till ett C#-objekt, i detta fall .NET-objekt [16]. Ett sätt att deserialisera JSON som användes i detta projekt är att först skapa en klass med egenskaper som representerar JSON-egenskaperna. I denna klass skapas en asynkron metod för att hämta data från databasen, i denna metod anropas metoden *JsonConvert.DeserializeObject* för att deserialisera objektet (se figur 2.6). Metoden är asynkron för att kunna köra parallella trådar samtidigt. Alla andra egenskaper som inte representeras i klassen ignoreras.

```
return JsonConvert.DeserializeObject<List<Models.Contact>>(result);
```

Figur 2.6, en kodrad från projektet där ett objekt deserialiseras

Serialisering och deserialisering samverkar för att transformera/återskapa dataobjekt till/från ett JSON-dokument.

2.9 HTTP metoder

HTTP står för Hypertext Transfer Protocol. Syftet med detta protokoll är att möjliggöra kommunikationen mellan klienter och servrar genom ett begäran-svar-protokoll. HTTP-förfrågningar är meddelanden som skickas från klienten för att initiera en åtgärd på servern. Dessa förfrågningar börjar med bland annat följande HTTP-metoder [17]:

- **GET:** GET-metoden begär en representation av den angivna resursen. Förfrågningar som använder GET bör endast hämta data utan att modifiera den.
- **POST:** POST-metoden används för att skicka data till en server för att skapa eller uppdatera en resurs med den angivna data som skickas med.
- **PUT:** PUT-metoden uppdaterar och ersätter befintliga resursdata med data som skickas med.

Dessa metoder har använts i projektet för att kunna ha en klient-server kommunikation. Klienten i applikationen skickar en förfråga till backend (databasen) om data som önskas representeras i applikationen eller ändras i databasen.

2.10 Request-URI

Kommunikation mellan en klient som använder mobilapplikationen Pindle och en webbserver sker med hjälp av ett Request-URI som står för Uniform Resource Identifier som identifierar den resurs på vilken begäran ska tillämpas. URI består av en sekvens med tecken som skiljer en resurs från en annan [18]. Allt den behöver innehålla är ett schemanamn och filsökväg (se figur 2.7). Schemat identifierar protokollet som ska användas för att komma åt en resurs på internet som till exempel HTTP eller HTTPS [19].

```
var requestUri = "https://localhost:5006/Contacts";
```

Figur 2.7, en rad kod som visar hur en request-URI kan se ut i frontend

2.11 SignalR

SignalR är en programvara med öppen källkod till ASP.NET som möjliggör funktionaliteten att skicka meddelande i realtid mellan flera klienter [20]. Det är alltså data som finns på serversidan som skickas till anslutna klienter i realtid. Liksom ASP.NET, har SignalR en högprestanda och ses som ett utav det snabbaste realtidsramverket. SignalR stödjer bland annat SQL server som kan användas för att till exempel skicka meddelanden mellan klienter. Med hjälp av ett API, kan klient-server designmönstret uppfyllas. Detta API är sedan anslutet till en databas så att meddelanden skall nås ut till alla klienter.

3. Metod

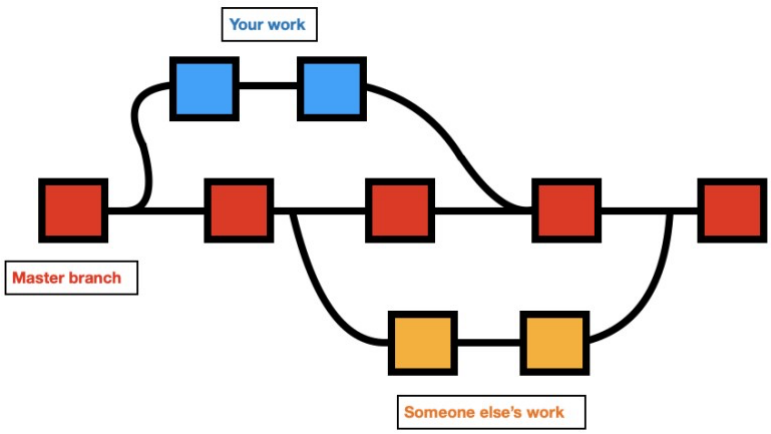
Innan projektets start diskuterades vilket som skulle vara det lämpligaste arbetssättet att arbeta utifrån men också vilka program som skall användas för att enkelt kunna arbeta på samma projekt samtidigt. Detta avsnitt går igenom de verktyg och utvecklingsmodeller som användes under projektets gång.

3.1 Visual Studio

Visual Studio är en avancerad programutvecklingsmiljö (IDE) utvecklat av Microsoft i syfte att utveckla bland annat webbsidor och mobilapplikationer till både Android och iOS. All kod som implementerades skrevs i Visual Studio. Några av de inbyggda programmeringsspråken som finns är Visual Basic .NET, C#, C++ och XML. Visual Studio använder sig av en kodredigerare som stödjer IntelliSense, en kodkomplettering-funktion som ska påskynda processen med att koda. Detta görs med hjälp av bland annat frågetips relaterade till syntaxfel, syntaxmarkering och automatisk indragning. Tack vare dess intuitiva kortkommandon, automatiska kompletteringar och felsökare är Visual Studio en enkel programutvecklingsmiljö att navigera i samt organisera sin kod [21].

3.2 Git

Git användes i detta projekt för att dela kod mellan gruppmedlemmarna. Git är ett distribuerat versionskontrollsystem med öppen källkod i syfte att låta flera utvecklare arbeta på ett och samma projekt. Genom att snabbt och effektivt dela filer, uppdateringar och andra projektrelaterade data mellan utvecklare skapas en mångsidighet i arbetsflödet. Med hjälp av gits funktion att kunna arbeta på flera branscher baserade på en master branch, där grunden av all källkod finns, kan flera utvecklare arbeta på olika delar av ett projekt samtidigt (se figur 3.1). Andra funktionaliteter som stödjer Git är att kunna ladda upp lokala ändringar (push), uppdatera befintliga filer (commit) samt ladda ned uppdaterade filer (pull) [22].



Figur 3.1, en illustration som visar hur Git funkar där två personer, blå och orange, arbetar parallellt på ett och samma projekt och kan pusha sin kod till master branschen av projektet.

3.3 Docker

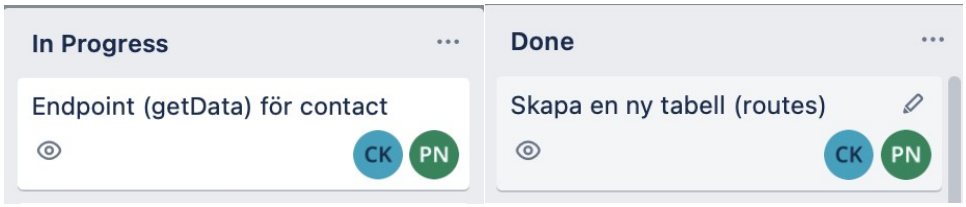
Docker är en öppen källkodsplattform, främst för macOS, för att bygga, distribuera och hantera kod i så kallade behållare. Plattformen stödjer designmönstret klient-server. En behållare är en programvara som paketerar kod och dess beroende av annat så att applikationen kan köras snabbt och tillförlitligt från en datormiljö till en annan. Syftet med Docker är att kunna ladda upp källkod på ett moln som man sedan har tillgång till när och var som helst. Detta gör det möjligt för utvecklare att köra koden enklare och säkrare med enkla kommandon genom ett enda API. Detta verktyg har blivit alltmer användbart i samband med att man går över till en molnbaserad utveckling. Docker kommer endast att användas vid körning av projektet på macOS [23].

3.4 Azure DevOps

Azure DevOps är en uppsättning samarbetsverktyg för programutveckling som låter korsfunktionella team arbeta med projekt av olika storlekar [24]. Med andra ord hjälper Azure DevOps utvecklare och teamet att skapa och distribuera applikationer oavsett språk, plattform eller moln. Eftersom företaget Elicit använder Azure DevOps till deras projekt användes det även i detta projekt i syfte att dela kod samt få möjligheten att spåra utvecklat arbete med teamet kontinuerligt med hjälp av Git.

3.5 Trello

Trello är en typ av scrumtavla som användes för att visuellt organisera arbetsuppgifterna (se figur 3.2). Tavlan är uppdelad i fem kolumner vars rubriker är “Sprint Backlog”, “Todo”, “In Progress”, “Testing”, “Code Review” och “Done”. I “Sprint backlog” skapades olika kort där varje kort innehöll en arbetsuppgift. Sedan förflyttades detta kort genom varje kolumn beroende på vart i processen man låg. Detta underlättade för gruppen att hålla koll på varje veckas uppgifter, vilka uppgifter som har gjorts tidigare samt vart i processen man var under veckan.



Figur 3.2, en illustration över två kolumner i Trello tavlan där In Progress visar vad som arbetas med just nu och Done visar vad som är färdigt

3.6 Agil Systemutveckling

Agil systemutveckling är en metodik som har vuxit fram ur mjukvaruindustrin som idag är en av det främsta arbetssättet som tillämpas. Syftet med att arbeta agilt är att flera ska kunna arbeta samtidigt på ett och samma projekt utan att det blir rörligt. Detta innefattar korta iterativa cykler där man förbättrar, analyserar och testar arbetsuppgifterna [25].

Arbetsprocessen i detta projekt inkluderade en veckas intervaller där man varje vecka hade ett möte tillsammans med produktägaren och planerade veckans arbetsuppgifter i Trello. Programutvecklingsmetoden som användes under projektets gång är parprogrammering där två utvecklare arbetar tillsammans vid en gemensam dator. Tekniken går ut på att den ena skriver kod medan den andra granskar varje kodrad som skrivs in. Rollerna växlas sedan under projektet så att både får skriva samt granska kod [26].

3.7 CRUD

CRUD (”Create”, ”Read”, ”Update”, ”Delete”) är en förkortning för skapa, läsa, uppdatera, radera. Dessa fyra operationer är grundläggande i programmering för att skapa en beständig lagring [27]. En sådan lagring innefattar all typ av datalagring 4 som håller igång lagringen trots att en enhet har stängt av. CRUD används framförallt i relationsdatabaser samt i databashanterare som exempelvis SQLite. Dessa fyra operationer anropas för att utföra operationen på en utvald data i databasen. Operationerna utförs av utvecklaren i form av kod eller genom ett grafiskt användargränssnitt i en databashanterare.

- **Skapa**: en operation som tillåter användaren att skapa en post i databasen. En post skapas genom att användaren fyller i all data som motsvarar de kolumner en tabell innehåller. Därefter skapas en rad i databasen med den data som fylldes i.

- **Läsa**: en operation som tillåter användaren att söka i databasen. Den tillåter användaren att hämta posts som användaren specificerar och därefter läser deras värde.

- **Uppdatera**: en operation som tillåter användaren att modifiera befintliga posts i databasen. Det kan vara att man vill ändra på några kolumner i databasen och behöver därmed inte ta bort hela tabellen och göra om.

- **Radera**: en operation som tillåter användaren att ta bort en hel post från databasen som inte längre behövs.

3.8 Swagger

Swagger är en uppsättning verktyg med öppen källkod som har i syfte att hjälpa utvecklaren att designa, bygga, dokumentera och använda REST API:er. Det tillåter både datorer och människor att förstå kapaciteten hos ett REST API utan direkt tillgång till källkoden [28]. Swagger är ett webbaserat användargränssnitt som därmed visar information om API:et visuellt. Man kan även testa API:ets endpoints genom HTTP GET och PUT metoder. Alltså ska det gå att genom webbsidan lägga till kontakter respektive rutter i applikationen. Se figur 1 i bilagor för att få en överblick över Swagger och dess funktionaliteter.

Swagger underlättade i arbetsprocessen på det sättet att databasens uppkoppling kunde kontrolleras via HTTP metoderna. Ett 500 error code Internal Server Error innebär att det är något problem med anslutningen till databasen. En anledning, främst för macOS, kan vara att Docker inte är igång. En annan anledning kan vara att man kör mot en annan server än den som har specificerats i databasen, dvs att man exempelvis kör mot en annan dators IP-adress och inte mot sin egen dators IP-adress “localhost”. Ett 200 code OK innebär att databasen är fungerande och det går att hämta och skriva till databasen via HTTP metoderna PUT och GET.

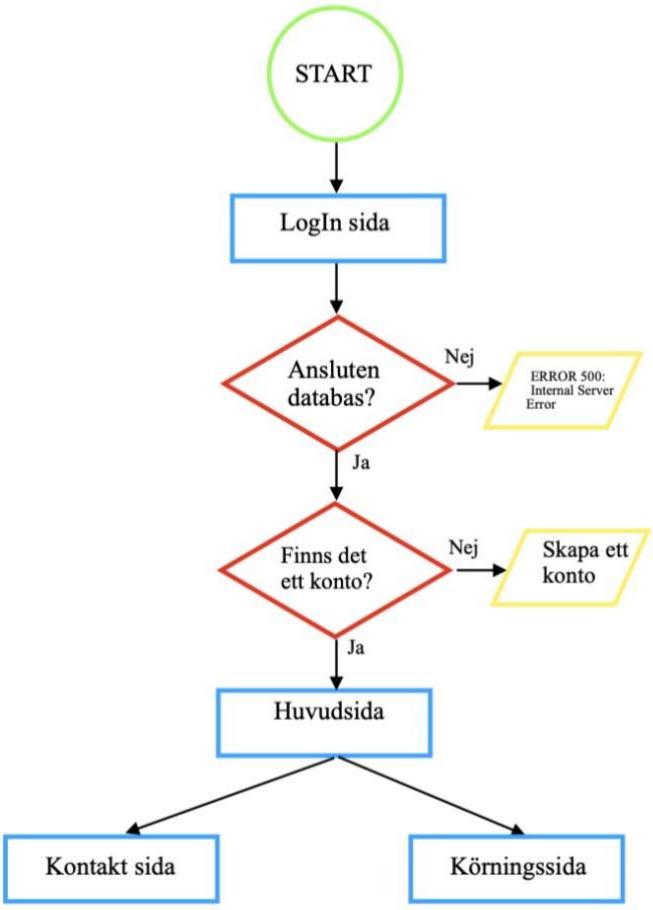
4. Genomförande

Detta avsnitt beskriver arbetsprocessen från en enkel databas till en utvecklad databas till prototypen Pindle. Uppsättningen av databasen på både macOS och Windows, tabeller och andra viktiga funktioner tas även upp.

4.1 Databasdesign

Innan kod började skrivas planerades en design för databasens struktur och begränsningar. Eftersom det blir svårare och svårare att göra ändringar när man väl är igång med utvecklingen av databasen krävdes det att gå igenom en process med flera steg [29]. Detta för att planera en så strukturerad och lättförståelig databas som möjligt.

Det första steget i en sådan process var att förstå logiken bakom applikationen och dess funktioner för att få en förståelse över hur databasen skall integrera med applikationen. Den viktigaste frågan som ställdes och som behövde ett svar innan implementeringen var "varför behövs en databas?". Svaret på den frågan var att en del data i applikationen behöver lagras för att vid senare tillfälle kunna hämta de. Ett så kallad flödesschema skapades för att visuellt se integrationen mellan applikationens funktionaliteter och databasen (se figur 4.1). Nedan finns flödesschemat över hur en alternativ väg kan se ut i applikationen och hur den interagerar med databasen. Vid inloggningen skall databasen se till att användaren loggar in med en befintlig mejladress, dvs ett mejl som redan finns sparat i databasen, annars krävs det att man skapar ett konto. Därefter kommer man till huvudsidan där man väljer om man vill gå till kontakt sidan eller körningssidan.



Figur 4.1, ett flödesschema som visar flödet från när man startar applikationen tills man väljer att gå till antingen kontakt sidan eller körningssidan.

När detta var tydligt började en planering av tabeller och relationer att diskuteras. Vilka tabeller behövs för att lagra all data samt hur ska dessa tabeller förhålla sig till varandra om sådant. Slutligen, planerades vilka endpoints som behövs implementeras för att uppfylla applikationens funktionaliteter. Det var främst två funktionaliteter som diskuterades vilket var skapandet respektive lagringen av kontakter och rutter.

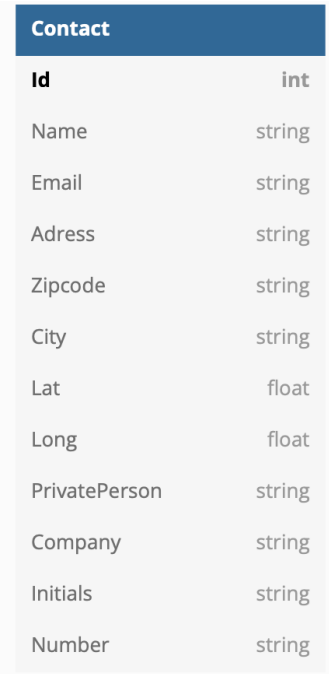
4.1.1 Tabeller i databasen

All data som lagras i databasen finns uppdelad i två tabeller där varje tabell lagrar diverse data. Varje tabell består av ett antal kolumner där vardera kolumn sparar en viss data. Två DTO-objekt skapades till respektive tabell som skulle ta hand om transporten av data mellan databasen och backend. Följande sektion presenterar de två tabellerna samt deras förhållande till varandra i form av ett ER-diagram.

4.1.1.1 Contact

En *Contact* tabell skapades för att kunna spara alla personuppgifter hos en användare. Tabellen består av ett antal kolumner bland annat ett namn, en adress och en mejladress (se figur 4.2). Kolumnerna sparas som variabler med olika datatyper. En användares Id sattes som en primärnyckel, dvs är den unik för alla användare. Användarens Id används inte direkt men finns som en kolumn i tabellen eftersom den skapas automatiskt i SQLite vilket gjorde att den behövde vara en primärnyckel. Den ideala lösningen är att ha användarens mejladress som primärnyckel och inte Id då man behöver logga in för att kunna använda applikationen. Det är även med den mejladressen som man loggar in med som identifierar varje användare.

All data som skickas fram och tillbaka mellan databasen och backend skickas i form av DTO-objekt. Två olika DTO-objekt skapades manuellt för *Contact* tabellen. Genom att lägga två nya filer i Visual Studio och döpa de till *ContactDTO.cs* respektive *CreateContactDTO.cs. ContactDTO* skapades för att skicka en användares data från backend till databasen. Det innebär att det objektet ser exakt likadant ut, dvs har identiska kolumner som *Contact* tabellen. *CreateContactDTO* skapades för att spara information som en användare skriver in vid skapandet av en ny kontakt i applikationen. Det objektet har alla kolumner som *Contact* tabellen har exklusive Id kolumnen då det får ett värde i SQLite automatiskt. Dessa DTO-objekt skickas mellan backend och databasen i form av migrationer. När en migration har skapats i backend innebär det att transporten av data från backend till databasen har gått igenom. En uppdatering av databasen krävdes för att kunna se den nya data i databashanteraren visuellt.

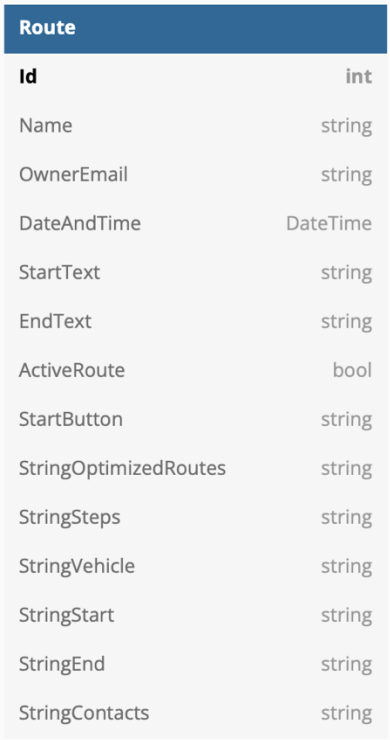


Figur 4.2, Contact tabellen i databasen

4.1.1.2 Route

För att kunna spara all information om en rutt skapades en *Route* tabell. Tabellen består av ett antal kolumner varav ett id, användarens mejladress och en destinationsadress är några exempel (se figur 4.3). Likaså här specificerades alla kolumner med datatyper. När en rutt skapas, sparas rutten med den mejladressen som användaren loggade in med. På så sätt kunde endast användarens egna rutter, dvs de rutter som man själv skapat, eller de rutter som man själv är en kontakt i hämtas. Detta för att inte en användare ska få alla rutter som finns sparade i databasen. Varje rutt identifieras med en rutt Id som är primärnyckel och alltså är unik för alla rutter.

För att spara en rutts information i databasen skapades även här två DTO-objekt som skickades mellan backend och databasen. *RouteDTO* och *CreateRouteDTO* som fungerar på samma sätt som *Contact* tabellens två DTO-objekt.



Figur 4.3, Route tabellen i databasen

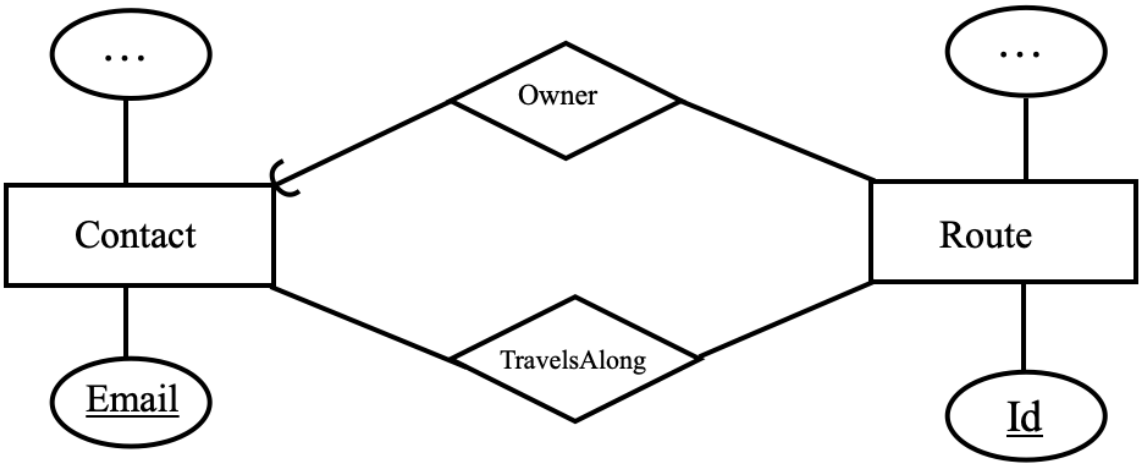
4.1.1.3 ER-diagram

ER-diagrammet illustrerar förhållandet mellan de två tabellerna *Contact* och *Route*. Tabellerna visas som entiteter, vilket är fyrkanterna. Varje tabell har ett antal kolumner som i ER-diagrammet visas som cirklar med punkter som är anslutna till entiteterna. Relationen mellan tabellerna visas i form av diamanter (se figur 4.4).

Relationen mellan en användare, dvs en kontakt, och en rutt är att varje rutt ägs av en kontakt. Varje rutt kan endast ägas av en användare vilket visas i diagrammet som en rund pil. Detta innebär även att det motsatta hållet inte stämmer, dvs att en rutt inte kan ägas av flera användare. Denna relation är i dagsläget implementerat manuellt i frontend och har därmed ingen mejladress som primärnyckel i *Contact* tabellen i databasen. Den ideala lösningen, som även ER-diagrammet nedan visar, är att tabellerna ska ha en främmande nyckel, på engelska “Foreign Key”, för att ta hänsyn till relationen mellan tabellerna i databasen och inte i frontend. Detta sker genom att tabellen *Contact* ska ha en

mejladress som ska vara en primärnyckel. På det sättet tydliggör det att en rutt-ägares mejladress är en användares mejladress.

En annan relation som ER-diagrammet illustrerar är att varje rutt har ett antal användare som ska vara med i rutten. En rutt kan ha hur många användare som helst och en användare kan vara med i hur många rutter som helst. Detta visas i ER-diagrammet genom att inga pilar finns mellan entiteterna och relationen *TravelsAlong*.



Figur 4.4, en illustration över ER-diagrammet

4.1.2 Modeller i frontend

En del data behövdes endast användas i frontend och var därför inte nödvändig att sparas i databasen. För att på ett säkert sätt integrera API:et med databasen skapades modeller i form av klasser. Två identiska modeller skapades i frontend av *Contact* respektive *Route* tabellerna i databasen. Utöver det skapades två modeller som i denna sektion skall presenteras.

4.1.2.1 Contact

Contact är en modell som är identisk, dvs har samma egenskaper, som tabellen *Contact* i databasen. Detta för att möjliggöra serialisering och deserialisering av data mellan applikationen och databasen. Det var även nödvändigt att kolumnerna i modellen hade samma egenskaper som kolumnerna i databasen för att kunna modifiera data.

4.1.2.2 Route

Route är en modell som är identisk med *Route* tabellen i databasen. På samma sätt som *Contact* modellen, är *Route* modellen nödvändig för serialisering och deserialisering av data från och till databasen. Till skillnad från *Contact* modellen, behövde *Route* modellen innehålla fler kolumner än *Route* tabellen i databasen för att hantera en del data i frontend som inte behövde sparas i databasen. Eftersom kolumnerna behöver vara identiska med det som finns i databasen, krävdes det att kolumnerna ignoreras vid serialiseringen och deserialisering av modellen. Dessa kolumner ignorerades med ett attribut, [JsonIgnore] (se figur 4.5).

```
[JsonIgnore]
public List<Contact> Contacts { get; set; }
```

Figur 4.5, en kodrad som visar användningen av JsonIgnore

Kolumner som ignorerades vid serialisering och deserialisering var främst de kolumner med andra datatyper än *int, bool, float* och *string*. Detta eftersom kolumnerna i databasen endast har de nämnda datatyperna. Detta betyder att alla andra kolumner i modellerna som hade en annorlunda datatyp behövde deserialiseras separat till en motsvarande kolumn av datatypen *string* i databasen. Projektgruppen tog beslutet att endast spara de variablerna med de ovannämnda datatyperna tillsammans med handledaren i företaget för att det var det smidigaste sättet att spara data. Exempel på variablers datatyper som inte kunde sparas i databasen är *Steps* som är en instans av klassen *Step[]* som i sin tur är en lista med alla steg i en rutt. Antal steg i en rutt är antal klienter som ska bli upphämtade. Denna kolumn sparas i databasen som en variabel av datatypen *string* och för att sedan kunna användas som *Step[]* i modellen krävs det en deserialisering (se figur 4.6 & 4.7).

```
[JsonIgnore]
public Step[] Steps { get; set; }
```

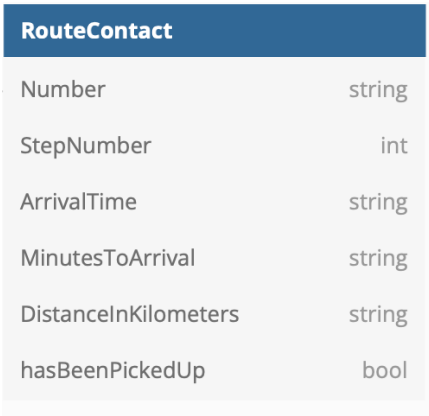
Figur 4.6, exempel på en kolumn av datatypen Step[] som inte sparas i databasen

```
[JsonProperty("stringSteps")]
public string StringSteps { get; set; }
```

Figur 4.7, exempel på en kolumn av datatypen Step[] sparas som StringSteps av typen string i databasen

4.1.2.3 RouteContact

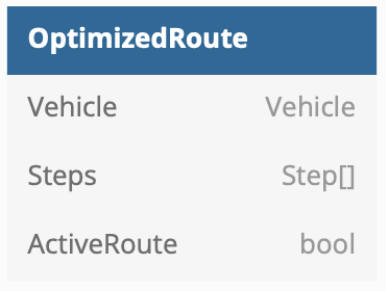
Varje rutt har en lista med användare som ska bli upphämtade. Detta implementeras med hjälp av en modell i frontend där *RouteContact* ärver från *Contact* modellen. Detta innebär att *RouteContact* har alla kolumner som *Contact* har samt några till som specificerar användarens plats i listan i en rutt (se figur 4.8). Denna modell finns inte med som en tabell i databasen vilket gör att den informationen inte sparas utan endast finns i frontend. Detta innebär även att det inte finns DTO-objekt till denna modell då endast tabeller som finns i databasen har tillhörande DTO-objekt.



Figur 4.8, RouteContact modellen i frontend som endast visar de extra kolumnerna som inte ärvs av Contact tabellen

4.1.2.4 OptimizedRoute

OptimizedRoute är en modell i frontend som innehåller all grundläggande data för en rutts optimering, dvs bland annat vilket fordon rutten skall köras med och hur många användare som finns med i rutten i form av antal platser som fordonet skall besöka (se figur 4.9). Även denna modell finns inte med i databasen som en tabell vilket gör att denna information inte sparas. Detta för att denna data endast behövs temporärt vid optimeringen av en rutt i frontend.



Figur 4.9, OptimizedRoute modellen i frontend

4.2 Uppsättning av databasen för macOS

För att sätta upp en enkel databas på operativsystemet macOS krävdes det att man först anordnade en inloggning med ett användarnamn och lösenord i en programutvecklingsmiljö, i detta fall Visual Studio. Denna känsliga data skulle inte kodas i källkoden, utan den sparades i en fil, *user-secrets*, gömd i datorn. Eftersom filen innehåller en känslig data krävdes det kommando för att både skapa filen och för att öppna och skriva i den. Med hjälp av kommandon som ramverket .NET Core erbjuder i Visual Studio, kunde denna fil på ett säkert sätt skapas och en inloggning implementerades (se figur 4.10). Samma inloggningsuppgifter användes sedan vid skapandet av en behållare i Docker. En behållare i Docker skapades för att paketera all kod som tillhör applikationen. Detta gör att det blir enklare att köra koden i olika utvecklingsmiljöer, exempelvis om man vill köra databasen lokalt eller mot en annan server.

```
dotnet user-secrets set "DbUser" "sa"
```

Figur 4.10, visar kommandot för att skapa ett användarnamn i user-secrets filen i terminalen i Visual Studio

Databashanteraren DBeaver ansluts sedan till databasen, i detta fall som finns på IP-adressen 127.0.0.1, som refererar till datorns lokala IP-adress eftersom databasen låg lokalt på varje dator. Docker behöver vara igång samtidigt för att anslutningen till databasen skall beviljas. För att sedan sammankoppla Visual Studio och databashanteraren krävdes det att man skapade en "appsettings.local.json" fil i Visual Studio där ett uttryck som kallas "ConnectionStrings" implementerades. Denna anslutningssträng specificerar information om en datakälla, dvs datahanteraren med dess IP-adress, och därmed tillät en anslutning till den (se figur 4.11). Därefter fick man databasen som var kopplad till Visual Studio.

```
{
  "ConnectionStrings": {
    "PindleDb": "Server=127.0.0.1;Database=PindleDb;MultipleActiveResultSets=true;"
  },
```

Figur 4.11, visar kodrader från backend som visar hur anslutningen skedde mellan databasen och applikationen i Visual Studio på mac

4.3 Uppsättning av databasen för Windows

Det skiljer sig i viss omfattning mellan uppsättningen av en databas på operativsystemen macOS och Windows. *Secrets.json* är en motsvarande fil av *user-secrets* som användes på macOS. Denna fil finns dock gömd direkt i Visual Studio och kräver inga kommandon för att skapa eller öppna den. Där skapades en inloggning som sedan användes i databashanteraren SQL Server Management Studio (se figur 4.12).

```
{
  "DbUser": "..",
  "DbPassword": ".."
}
```

Figur 4.12, visar secret.json fil där ett inlogg med användarnamn och lösenord skapades

På samma sätt som för macOS, skapades även i Windows en "appsettings.local.json" där "ConnectionStrings" implementerades som tillät en anslutning mellan datorns server och databashanteraren (se figur 4.13). Skillnaden här är att i Windows kunde man ansluta direkt till datorns server medan på macOS måste man skapa en behållare i Docker och ansluta databashanteraren till den. På macOS behövde en anslutning ske till datorns lokala IP-adress medan på Windows behövde man bara skriva datorns namn som refererar till datorns lokala IP-adress.

```
"ConnectionStrings": {
  "PindleDb": "Server=CynthiasDator\\SQLEXPRESS;Database=PindleDb;MultipleActiveResultSets=true;Integrated Security=true"
},
```

Figur 4.13, visar kodrader från backend som visar hur anslutningen skedde mellan databasen och applikationen i Visual Studio på Windows

4.4 Endpoints

Som tidigare nämnts i sektion 3.4, fungerar API på så sätt att den skickar förfrågningar och får en respons från en webbserver eller webbläsare. Platsen där API skickar förfrågningar och där databasen finns kallas för endpoints. Endpoints används för att skapa specifika, effektiva och komplexa lösningar som bemöter ett specifikt behov [30]. I detta projekt har olika endpoints implementerats för att uppfylla applikationens olika funktionaliteter.

- **GET**: För att kunna skapa en endpoint som ska visa alla kontakter eller rutter användes HTTP metoden GET som begär data från servern. Som tidigare nämnt i sektion 3.9, förfrågningar som använder GET metoden bör endast hämta data från servern och har absolut ingen möjlighet att modifiera datan [31]. I projektet implementerades GET-förfråga metoden med den angivna URI som en asynkron operation och har namngetts *GetData* i klassen *Api.service* som ger ett svar när en begäran skickas till API. *GetData* användes i bland annat *GetContact* metoden i *Api.service* för att hämta alla kontakter som är sparade i databasen. Genom ett anrop till metoden *GetData* med ett URI som parametern hämtas data som ett JSON-dokument som i sin tur behöver skrivas om till .NET-objekt. Översättningen från ett JSON-dokument till en sträng text sker genom deserialisering av dokumentet.

  Metoden *GetMyRoutes* hämtar användarens egna rutter, dvs rutter som är skapade av användaren men även de som användaren är en passagerare i. I denna metod anropas metoden *GetData* med syfte att hämta rutter från databasen med hjälp av HTTP-GET metod. På samma sätt som kontakter, hämtas rutter från databasen genom deserialisering.

- **PUT**: För att kunna skapa ett endpoint som ska spara alla kontakter och rutter i databasen användes HTTP metoden PUT som skapar eller ersätter befintliga data i servern med nya data [32]. Förfrågningar som använder PUT bör kunna uppdatera sparade tabellerna i databasen med en ny rad. I projektet implementerades PUT-förfråga metoden med angivna data och URI som en asynkron operation och har namngett *PutData* i *Api.service*. *PutData* metoden anropas med ett objekt och URI som parametrar i bland annat *SaveContact* i *Api.service* för att spara alla kontakter och rutter i databasen. Objektet som skickas in med som parameter till *PutData* metoden bör serialiseras alltså skrivas om till sträng i formen av ett JSON-objekt.

  Metoden *SaveRoute* sparar rutter med angiven mejladress i databasen som ett JSON-objekt. Som tidigare nämnts i sektion 4.1.1.2, är mejladressen vid skapandet av en rutt densamma som den mejladressen användaren loggar in med. I denna metod anropas metoden *PutData* för att spara en rutt i databasen med hjälp av ett HTTP-PUT metod. Därmed serialiseras objektet och sparas i databasen.

- **POST**: I prototypen som gruppen fick från början, fanns en *PostData* metod som använder HTTP metoden POST. *PostData* metoden används för att lägga till nya data. Denna metod anropas i bland annat metoden *GetId* där syftet är att skapa nya id för rutter efter de optimeras. Metoden varken hämtar eller sparar värde i databasen, utan bara finns i frontend av projektet.

4.5 Felhantering

Detta delavsnitt presenterar hur gruppen valde att felhantera eventuella errors och fördelarna med de tagna besluten.

4.5.1 Inloggningssida & felhantering

I detta projekt har en inloggningssida implementerats för att användarna ska kunna skapa ett konto och logga in. Utseendet och gränssnittet implementerades av den andra samarbete gruppen, grupp 22, medan logiken implementerades av paret som skriver denna rapport. Tillsammans med handledaren bestämde projektgruppen att skapa varje användare som en kontakt, alltså ingen ny tabell för användaren har lagt till. Alla användaren räknas som kontakter i detta projekt och kan därmed använda alla metoder och egenskaper som en kontakt har, vilket också betyder att en användare kan skapa ett konto för en annan användare genom att lägga till denna användare som en kontakt i applikationen. Projektgruppen valde att göra på det sättet eftersom inga nya egenskaper krävs för en användare och därmed blir koden förståelig och enkel att läsa av samtidigt som den uppfyller alla krav för funktionalitet.

Logiken för denna sida låg i att kontrollera om mejladressen som man loggar in med finns redan sparad i databasen eller inte. Detta har implementerats genom att skapa en metod av datatypen boolean i vymodellen för inloggningssida som loopar igenom alla kontakter i databasen och jämföra med den mejladressen som skrivs i mejladress fält i inloggningssida. Projektgruppen valde att jämföra på det sättet eftersom det redan har implementerats en metod som hämtar alla kontakter, då behövdes det endast ett anrop till denna metod och ett if-sats. Ifall användaren inte finns registrerad i databasen, dyker det upp ett felmeddelande som informerar användaren att hen behöver först skapa en inloggning och hänvisar till registreringssida. För detta felmeddelande används metoden *DisplayAlert* där en sträng som innehåller informationen om felet skickas med (se figur 4.14). *DisplayAlert* är en av metoderna som Xamarin.Form har för att interagera med användaren via en popup [33]. Projektgruppen valde just *DisplayAlert* för att det var lämpligast och enklast att använda för just denna task.

```
else
{
    message = $"{message} \nEmail is not found, please sign up.";
    await Application.Current.MainPage.DisplayAlert("Alert!", message, "Ok");
}
```

Figur 4.14, felhantering för inloggningssida

Ifall användaren har inget registrerat konto i databasen, kan hen skapa ett konto i registreringssida där man får fylla i alla fält som krävs bland annat namn, mejladress, mobilnummer osv. På samma sätt som inloggningssida skapades, skapades även denna sida med hjälp av exjobbs gruppen 22. Felhantering implementerades på samma sätt som inloggningssida med en *DisplayAlert* där det kontrolleras om det finns redan en sparad användaren som har samma mejladress. Tanken är att mejladress ska vara den unika egenskapen som skiljer mellan alla användare

4.5.2 Null-hantering

Tabellerna i Pindle-backend skapades med variabler av olika datatyper som representerar kolumner i en tabell. Några av variablerna har en datatyp som följs av ett litet frågetecken bredvid (se figur 4.15). Att en variabel ska ha en datatyp följd av ett frågetecken i C# betyder att variabeln inte får vara null. Paret använde sig av detta sätt för null-hantering för att garantera att viktigaste variabler får ett värde vid exekveringen av koden.

```
public string? StartText { get; set; }

public string? EndText { get; set; }
```

Figur 4.15, Två kolumner av tabellen Route i backend

Några variabler skapades med ett attribut, [MaxLength], som begränsar storleken på variabeln (se figur 4.16). Detta attribut anger den maximala längden på datavärdet som tillåts för en egenskap, den används för egenskaper av datatyper *string* och *byte* [34]. Projektgruppen valde att använda attributet [MaxLength] på några kolumner för att inte behöva generera en stor plats i minnet som inte kommer att användas i projektet. Ett exempel på ett sådan kolumn i är variabeln *StringOptimizedRoutes* som har ett attribut på 20 000 vilket betyder att objektet får maximalt bestå av 20 000 tecken.

```
[MaxLength(20000)]
public string StringOptimizedRoutes { get; set; } = null!;
```

Figur 4.16, en variabel i Route tabellen som begränsas till längden 20 000

4.6 Automatiskt meddelande

Ett automatiskt meddelande behövde implementeras som skulle skickas självgående i applikationens chatt. Varje användare i applikationen är en klient och databasen är servern. Detta möjliggjorde att alla klienter i en chatt får ett meddelande samtidigt i realtid. Tanken med det automatiska meddelandet är att en användare skall meddela resten av användarna i chatten är den är framme hos en specifik användare. Detta ska ske genom att ett automatiskt meddelande skickas i chatten som talar om för de andra vart användaren är.

Med hjälp av ramverket SignalR och designmönstret klient-server kunde logiken för att kunna skicka samt ta emot meddelande mellan backend och applikationen genomföras. Den viktigaste kommunikationen mellan backend och alla dess klienter är att de är på en och samma hub. I denna sektion kommer de viktigaste delarna som gör kommunikationen möjlig att presenteras.

4.6.1 Implementation av Chathub i backend

ChatHub är den delen i backend som ger användaren tillåtelse att skicka och ta emot meddelanden. Syftet med ChatHub är att hantera server-klientkommunikationen mellan backend och dess klienter. I hubben implementerades en metod *SendMessage* som hanterar alla meddelanden som skickas till hubben och sedan skickar vidare meddelandet till alla klienter som lyssnar på en och samma hub (se figur 4.17).

```
public Task SendMessage(string routeId, string email, string message) =>
    Clients.Others.SendAsync("ReceiveMessage", routeId, email, message);
```

Figur 4.17, en skärmbild på SendMessage metoden i ChatHub

4.6.2 Implementation av Klient i applikationen

Integrationen mellan ChatHub och dess klienter sker i en service fil, *ChatService.cs*, som skapades i applikationen. Syftet med denna service är att hantera anslutningen mellan backend och klienterna samt skicka och ta emot meddelanden däremellan.

Anslutningen till ChatHub sker på samma sätt som när databasen skall anslutas till en server (se figur 4.18). Det är alltså möjligt att köra hubben både lokalt och mot en annan IP-adress.

```
_hubConnection = new HubConnectionBuilder()
    .WithUrl($"https://localhost:5006/chatHub"
```

Figur 4.18, en skärmbild från ChatService som visar hur man ansluter servern till ChatHub med hjälp av ett URI

För att kunna skicka ett meddelande till ChatHub implementerades en metod *SendMessage* i ChatService.cs. Syftet med metoden är att skicka ett meddelande via en nyckel som kommer att söka efter en metod i ChatHub i backend som matchar nyckeln. Detta för att hubben ska kunna ta emot meddelandet och sedan skicka vidare den till alla användare som aktivt lyssnar.

För att vidare kunna ta emot ett meddelande från ChatHub skapades en *On-metod* i ChatService. Syftet med denna metod är att bevara en nyckel som sedan jämförs med en nyckel i metoden *SendMessage.* Är dessa två nycklar identiska, kan ett meddelande skickas och tas emot via Hubben. I detta fall är nyckeln ”ReceiveMessage”. För att kunna avgöra om en klient aktivt lyssnar på ett meddelande som skickas till ChatHub, implementerades en *OnReceieveMessage* metod (se figur 4.19). Denna metod avgör vilka som ska få meddelanden beroende på om nycklarna är densamma.

```
_hubConnection.On<string, string, string>("ReceiveMessage", (routId, email, message) => OnReceiveMessgae(routId, email, message));
```

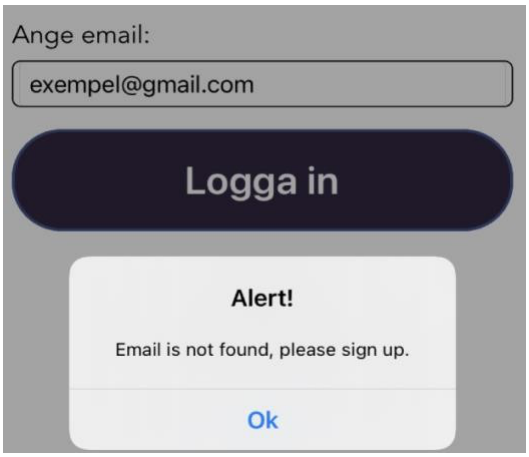
Figur 4.19, en skärmbild på on-metoden som implementerades som bevarar nyckeln

5. Resultat
Detta avsnitt summerar projektets resultat samt besvara frågeställningarna.

5.1 Slutresultatet
Projektet resulterade i en utvecklad databas för prototypen Pindle där funktionaliteter som att skapa ett konto samt spara sina kontakter och rutter. För att korrekt information skall sparas i databasen skapades två tabeller för kontakter respektive rutter med diverse kolumner. Liknande tabeller skapades även som modeller i frontend för att anknytningen mellan frontend och databasen ska gå igenom. Detta för att säkerställa att korrekt data skickas emellanåt samt sparas.

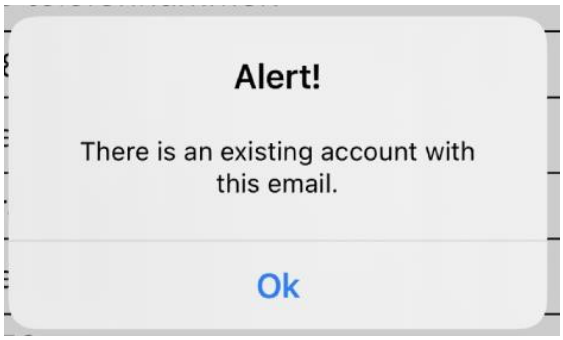
Databasen ska ligga på Elicits server vilket innebär att en URI länk behöver fås utav företaget för att kunna köra applikationen mot en gemensam databas och inte mot en lokal. Därför skall URI länken uppdateras med den nya från företaget i alla endpoints i frontend som är anslutna till databasen.

Efter att man har körande backend, dvs en fungerande databas, kan applikationen startas och man kommer till en inloggningssida. Där loggar man in med ett befintligt konto, dvs med en redan registrerad mejladress. Det är alltså en kontakt som finns sparad i databasen som applikationen är ute efter, annars dyker ett felmeddelande upp som meddelar användaren att kontot inte är registrerat (se figur 5.1).



Figur 5.1, en illustration över ett felmeddelande vid inloggningen

Om man inte har ett befintligt konto finns det möjligheten att skapa ett konto i registreringssidan. Där skall alla nödvändiga personuppgifter fyllas in. Skriver man in en mejladress som redan finns i databasen dyker ett felmeddelande upp som meddelar användaren att det redan finns ett konto med den mejladressen (se figur 5.2).



Figur 5.2, en illustration över ett felmeddelande vid skapandet av ett nytt konto

De två viktigaste funktionerna, som även är kärnan i projektet, är att kunna hämta och skapa nya kontakter samt rutter. Detta görs i två olika sidor i applikationen. Ena sidan innehåller användarens kontaktlista och den andra innehåller användarens rutter som den är med i eller har skapat själv. I båda sidorna har användaren även möjligheten att skapa en ny rutt respektive kontakt. Genom databasen i SQLite har dessa två funktionaliteter möjliggjorts, dvs att det både går att spara data och sedan hämta den data vid senare tillfälle. När en ny kontakt har blivit tillagd i kontaktlistan innebär det även att den då har blivit sparad i databasen med bland annat ett unikt id. Samma gäller för när en ny rutt läggs till, då sparas den med en unik rutt id.

Ett automatiskt meddelande implementerades även som skulle skickas självgående i chatten till alla användarna i en rutt när en specifik användare har kommit fram till vardera kontakten (se figur 5.3). Detta för att alla som är med i rutten skulle ha tillgång till den specifika användarens platsinfo. Detta gjordes med hjälp av SignalR som möjliggjorde kommunikationen när det gäller att skicka asynkrona meddelanden mellan olika klienter i applikationen.

Figur 5.3, en illustration över hur två automatiska meddelanden kan se ut i chatten

5.2 Resultatanalys

Svaret på frågeställningen “kan en sökbar databas i SQLite som integrerar med ett REST API i .NET Core utvecklas till mobilapplikation-prototypen Pindle?” är ja, det går att skapa en databas i SQLite som integrerar med ett REST API i .NET Core och är anpassad till Pindle. Med hjälp av de två tabellerna, Contact och Route, i databasen samt modellklasserna i frontend har hämtningen/lagringen av data kunnat genomföras. Dessutom användes olika endpoints för att integrationen mellan REST API:et och databasen skall fungera, dvs att data skickas och hämtas från databasen.

Underfrågeställningen ”skulle skapandet av endast två tabeller vara tillräcklig för en sådan databas?” kan besvaras med att ja, det är tillräckligt att endast skapa två tabeller i databasen för att kunna spara all nödvändiga data. I dagsläget används samma tabell i databasen (*Contact*) för både en användare som loggar in och en kontakt som senare kan läggas till av en användare i prototypen. Detta eftersom de har exakt samma egenskaper och därför behövs ingen ytterligare tabell skapas i databasen som skiljer mellan en användare och en kontakt.

Svaret på den andra frågeställningen ”kan ett automatisk meddelande som ska skickas i en gruppchatt och är anpassad efter prototypens chattsystem implementeras?” är att det går förutsatt att det implementeras i ramverket SignalR. Detta eftersom det redan finns ett chattsystem i den befintliga prototypen för Pindle utvecklat i SignalR. För att det automatiska meddelandet skall skickas i den befintliga chatten krävdes det att automatiska meddelandet skickas i samma hub som resterande meddelanden för att nå alla användare i chatten i realtid.

6. Diskussion
Detta avsnitt diskuterar syftet, frågeställning och resultatet i rapporten. Eventuella problem som projektgruppen bemötte under arbetstiden kommer även att presenteras och analyseras.

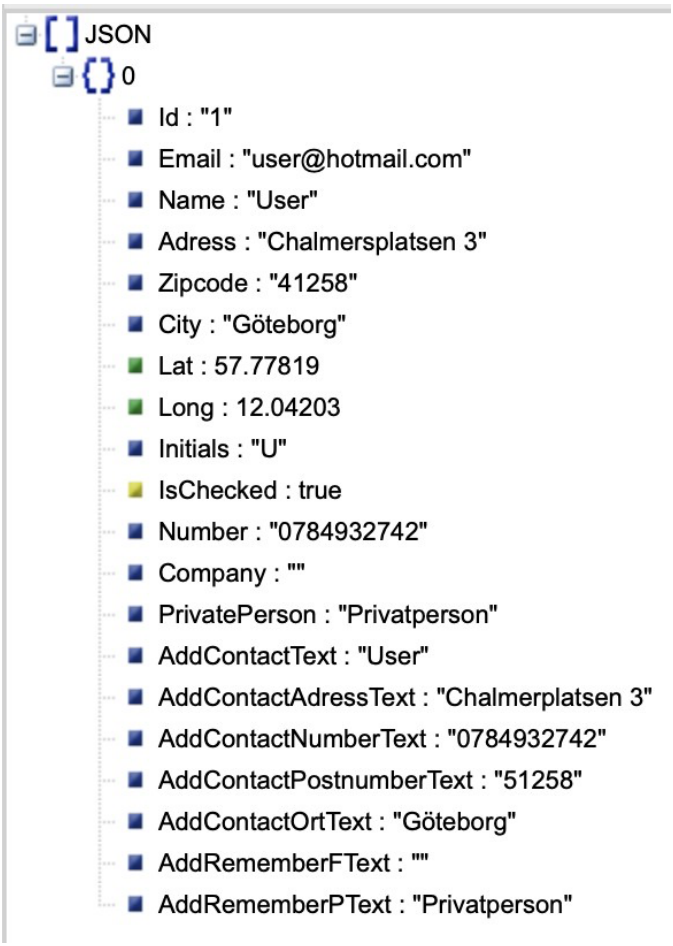
Resultatet visar att målet med projektet är uppnått då en fungerande och sökbar databas är färdigprogrammerad. Tack vare de tagna tekniska valen som förenklade hela arbetsprocessen men ändå uppfylla kraven för de olika funktionaliteter, kunde en databas och ett REST API i .NET Core för prototypen Pindle implementeras. De tekniska valen inkluderar databasdesign och all verktyg som användes. Agil systemutveckling och Trello var användbara verktyg och metoder inte bara för planering av projektet utan också för att underlätta arbetsflödet

6.1 Felsökning
Tekniken som användes mest för felsökning var debugging och brytpunkter. Debugging gjordes genom att sätta olika brytpunkter i koden med syfte att stoppa exekveringen av koden där det händer något som skulle kunna vara relevant för problemet. Därefter, körs koden steg för steg och genom att hovra över variablerna kan man se vilka värden de får i de olika steg. Projektgruppen valde att felsöka med hjälp av denna teknik för att enklare kunna se vart koden börjar skicka ut fel data och fel värde på variablerna. Denna teknik är att det är en ganska känd teknik för felsökning i programmering och därmed finns många bra tutorials på internet. Dessutom är tekniken lättlärd och har oftast kunnat visa felet i detta projekt. Men debugging har inte alltid lyckats hjälpa projektgruppen med felsökning av felet i programmet, och där vände sig paret till en annan teknik som kommer att förklaras i nästa stycke. Just i detta projekt kördes frontend och backend i två olika projekt i Visual Studio, vilket gjorde det svårare för paret att veta i vilket projekt brytpunkterna ska läggas. Ibland kunde felet ligga i backend projektet, men det skulle inte märkas förrän frontend har kört. Felsökning skulle då ha börjat i frontend tills att ett fel leder till det andra och paret märker att det felet faktiskt kan ligga hos backend, vilket uppenbarligen tog längre tid än förväntad för att hitta felet.

När brytpunkter och debugging inte skulle var tillräckliga för att hitta ett fel i programmet, använde sig projektgruppen av en hyfsat krävande metod som har lyckats leda till vart felet låg. Metoden är att med hjälp av de satta brytpunkter jämföra resultatet som fås av en fungerande gammal version av programmet i en gammal bransch med den nuvarande koden. Det togs olika skärmdumpar av de intressanta tabellerna för ett specifik fel i båda versioner av programmet och jämfördes. Ibland kunde datan kopieras, men det var i JSON-format vilket gjorde det svårt att läsa av de eftersökta variablerna (se figur 6.1). Online verktyget JsonViewer [35] användes för att enklare kunna se variablerna och dess värde (se figur 6.2)

```
[{"Id":"1","Email":"user@hotmail.com","Name":"User","Adress":"Chalmersplatsen 3",
"Zipcode":"41258","City":"Göteborg","Lat":57.77819,"Long":12.04203,"Initials":"U",
"IsChecked":true,"Number":"0784932742","Company":"","PrivatePerson":"Privatperson",
"AddContactText":"User","AddContactAdressText":"Chalmerplatsen 3",
"AddContactNumberText":"0784932742","AddContactPostnumberText":"51258",
"AddContactOrtText":"Göteborg","AddRememberFText":"","AddRememberPText":"Privatperson"}]
```


Figur 6.1, En kontakt från Contact tabellen i string-format



Figur 6.2, en kontakt från Contact tabellen i form av JSON-format

6.2 Sparandet av null i databasen

Under två veckors arbetstid försökte projektgruppen att lösa problemet med sparandet av null-värden i databasen. Problemet uppstod när projektparet försökte skicka de hämtade rutter från databasen till optimeringen. De rutter som hämtades från databasen hade null-värde i några av de viktigaste variabler som behövdes för optimeringen av en rutt, som till exempel variabeln som bestämmer vilket fordon rutten ska köras med. Projektgruppen undersökte och felsökte problemet med hjälp av debugging och jämförelse av det önskade resultatet och resultatet som fås. Efter långa dagar och med hjälp av handledaren Forsberg hittades den mest ultimata lösningen för detta fel. Lösningen var i form av ett foreach-loop som skulle gå igenom alla rutter och tilldela nya riktiga värde till de variablerna som

behövdes för optimeringen av rutten. De nya värde hämtas från tabellerna som endast finns i frontend genom ett URI till en av företagets tjänster för optimering av rutter, de sparas alltså inte i databasen. Denna loop skrevs i en metod i *Api.Service* som kallas för *GetOptimization* där all optimering sker.

6.3 För- och nackdelar med parprogrammering

Som tidigare nämnts i sektion 3.6 är parprogrammering en programutvecklingsteknik som låter två programmerare att jobba tillsammans på en dator, där den ena skriver kod men båda tänker och granskar koden som matas in. Projektgruppen valde att jobba utifrån denna teknik av många anledningar. Den första anledning är att den ena studenten hade inte tillräcklig stark dator för att kunna köra det tunga projektet, datorn är av typen *Lenovo ThinkPad X260* som har processorn *Intel Core i5* tillät inte körandet av Android emulatorn. Den andra anledning var att paret har tidigare arbetat med denna teknik i tidigare kurser, vilket underlättade arbetsflödet och även kommunikationen. Dessutom, har det visat sig enligt studier att programmerare som jobbar ihop producerar kortare och mer förståeliga program än de som jobbar ensamma [8]. Parprogrammerare får oftast jobbet klart mycket snabbare än en programmerare som får samma uppgift. Slutligen, valdes denna teknik eftersom kunskapen skulle förmedlas mellan paret och båda skulle lära sig från varandras misstag och knep.

Dock har det inte gått alltid lika strålande med att parprogrammera under hela arbetstiden. Paret behövde alltid planera i förväg hela veckan var och när skulle möten ta plats och tid. Vissa veckor var det svårt att hitta en tid där båda var tillgängliga, då det var en stressig period när det gäller det privata livet och studierna för paret. Därför var den ena studenten ibland tvungen att jobba under fritiden för att kunna passa den andras planer. De flesta möten togs via Zoom där det var enkel att skärmdela, men när paret behövde träffas på plats blev det lite svårt att båda skulle var lika aktiva på programmeringen och slutade med att bara den ena kunde se tydlig skärmen. Friheten i jobbet var alltså någorlunda begränsad med tanke på att hitta tid och plats som passar för paret.

6.4 Säkerhet

Säkerheten i REST API:et och databasen prioriterades inte av projektgruppen, då handledaren Forsberg förklarade tydlig från början av projektet att det är något som en annan ingenjör på företaget skulle fixa senare. Därför bestämde paret tillsammans med samarbetsteamet 22 och Forsberg, att inte fråga efter lösenord vid inloggningen då det visar tydlig att prototypen inte är tillräckligt säker. Under den näst sista arbetsveckan fick exjobbsgrupperna en tutorial av en ingenjör på företaget på hur olika filer bör läggas till för att öka säkerheten i databasen. Resultatet av den ökade säkerheten är att ingen skulle kunna ansluta till backend och REST API:et om inte de har nyckeln och lösenordet som har lagts till. På grund av tidsbrist kunde inte projektgruppen investera mer tid på att faktiskt förstå och analysera den säkerhetslösning som användes eller jämföra och testa andra sätt att säkra API:et.

6.5 Miljö- och etiska aspekter

Miljön påverkas alltid av nya teknologier och teknik som utvecklas och det är människors samt utvecklarens plikt att försöka minska de negativa effekter och skydda planeten. Miljön bör alltså alltid finnas i åtanke när en ingenjör utvecklar en viss produkt eller tjänst. Detta arbete gjordes för att minska bilutsläpp som förstör miljön och för att gynna användarens ekonomi. Lyckligtvis är allt som behövs för denna applikation prototyp är en smart mobiltelefon som de flesta människor i Sverige har och inga ytterligare fysiska komponenter krävs.

Utifrån olika etiska perspektiv och ramverk kan säkerheten i databasen diskuteras. Generellt sett, innehåller databaser i olika webbsidor och applikationer de mest känsliga data som till exempel kundernas personuppgifter. Därför är det viktigt att skydda databasen mot angrepp, datakorruption och missbruk. Detta projekt tog inte hänsyn till säkerheten i databasen eftersom det är en prototyp och det var bestämd att en annan ingenjör i företaget skulle fixa det senare. Därför var det viktig att inte ha ett lösenordsfält i inloggningssidan för att indikera att inga känsliga personuppgifter behöver lämnas. Men hade prototypen utvecklats till en applikation som ska publiceras då skulle projektgruppen lägga all fokus på säkerheten av sparad data i databasen. Det är också viktig att säkerställa att kommunikationen mellan databasen och applikationen är säker och inga avlyssnare stör vägen. Det är utvecklarna och företagets ansvar att se till att deras tjänst är säkra och att det finns inga sårbarheter de är medvetna om.

7. Slutsats och fortsatt arbete

Databaser är något som används dagligen utan att man visuellt ser det framför sig och kommer att användas betydligt mycket mer i samband med att allt börjar digitaliseras. En hel del kunskap har inhämtats under projektets gång i och med alla verktyg och utvecklingsmodeller som har använts. Arbetssättet under projektets gång har varit till nytta då det var tydliga arbetsuppgifter i Trello, begripliga och relativt enkla verktyg samt omfattande utvecklingsmodeller. Om ett liknande projekt skulle göras på nytt skulle det gå mycket smidigare och ytterligare funktioner skulle implementeras på grund av det nya tankesättet kring planering, arbetssättet samt design och struktur.

Sammanfattningsvis har projektets mål uppnåtts och med några hinder och avgränsningar har en utvecklad databas med ett REST API skapats med kärnan att göra databasen sökbar och effektivt kunna lagra data. Hinder såsom att spara null-värden i databasen var acceptabelt tills det hanterades vilket drog ut på tiden. Avgränsningar såsom att inte logga in med ett lösenord och enbart med en mejladress begränsades för att signalera för användaren att databasen inte är helt säker ännu.

Databasen har fortfarande luckor som går att vidareutveckla och förbättra för att uppfylla applikationens krav i framtiden. Några funktionaliteter i applikationen kommer att behöva implementeras för att applikationen ska bli mer användarvänlig. Ett exempel på vidareutveckling är en sida för privat chatt mellan alla användare där meddelanden och gruppchatten sparas i databasen. För detta, behövs ytterligare en tabell i databasen som sparar varje meddelande med sin ID, sändarens ID, mottagarens ID, texten som skrivs in och möjligtvis konversationens ID ifall det skulle vidareutvecklas till en gruppchatt. Dessutom, används inte mobilnummer i dagsläget som krävs från användaren att mata in till något syfte vilket kan anses vara onödigt. Mobilnummer skulle ha en stor betydelse i applikationen ifall man kunde ringa till varandra i chatten för att ha lättare kommunikation mellan förare och kunder.

I detta projekt var det lämpligast att använda Contact tabellen för användaren också. Men i framtiden borde användaren ha en egen tabell i databasen för att öka säkerheten och lägga till andra egenskaper ifall det skulle behövas. För användarens tabell, borde användarens ID, mejladressen, mobilnummer, adress och möjligtvis en checkbox för om användaren är en förare eller kund. Med ökad säkerhet i applikationen borde det även läggas till ett textfält i inloggningssida för lösenord, som i sin tur ska sparas i databasen i den framtida användare tabell. Lösenordet visar tydligt för användaren av applikationen att personuppgifterna kommer att vara skyddade. Slutligen, bör användaren av sunt förnuft kunna logga ut från applikationen och logga in med ett annat konto. Det gör applikationen mer användarvänlig och enklare att använda.

Referenser

[1] “Transportör - Ruttoptimering”, *pinDeliver*. [online]. Tillgänglig: https://pindeliver.com/sv/transport/ , Hämtad: april, 08, 2022

[2] M. Roos, “Kom igång med databasen SQLite”, *dbwebb*, Jan, 09, 2017. [online]. Tillgänglig: https://dbwebb.se/kunskap/kom-igang-med-databasen-sqlite , Hämtad: april, 08, 2022

[3] “Vad är C#?”, *Limetta*. [online]. Tillgänglig: https://limetta.se/tips-metoder-for-digitalaprojekt/Vad-ar-csharp/?gclid=CjwKCAjw7IeUBhBbEiwADhiEMecakmHwDpiJiJRb2wBVW9Rz9LtWdZ2LM_6S4J_cxFDP460Fh8q0jxoCckwQAvD_BwE , Hämtad: april, 25, 2022

[4] "The model-view-viewmodel pattern", *Microsoft*, Aug, 07, 2021. [Online]. Tillgänglig: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm , Hämtad: april, 26, 2022

[5] “What Is a Database?”, *Oracle*, 2021. [online]. Tillgänglig: https://www.oracle.com/database/what-is-database/ , Hämtad: april, 26, 2022

[6] “Why use a database”, *futurelearn*, [online]. Tillgänglig: https://www.futurelearn.com/info/courses/introduction-to-databases-and-%20sql/0/steps/75849 , Hämtad: april, 30, 2022

[7] “Vad är ett databashanteringssystem (DBMS)”, *Livetstrad.com*, Juni, 26, 2021. [online]. Tillgänglig: https://livetstrad.com/vad-ar-ett-databashanteringssystem-dbms/ , Hämtad: april, 30, 2022

[8] “About SQLite”, *sqlite.org*. [online]. Tillgänglig: https://www.sqlite.org/about.html , Hämtad: maj, 02, 2022

[9] H. Friedman, “Top 5 API Integration Platforms: Features and Cost Breakdown”, *improvado.io*, Juli, 20, 2021. [online]. Tillgänglig: https://improvado.io/blog/the-best-api-integration-platforms , Hämtad: maj, 02, 2022

[10] “Data Transfer Object DTO Definition and Usage”, *okta.com*. [online]. Tillgänglig: https://www.okta.com/identity-101/dto/ , Hämtad: maj, 02, 2022

[11] “Database migration: Concepts and principles (Part 1)”, *cloud.google.com*, Maj, 11, 2020. [online]. Tillgänglig: https://cloud.google.com/architecture/database-migration-concepts-principlespart-1 , Hämtad: maj, 02, 2022

[12] “.NET Core Overview”, *tutorialsteacher.com*. [online]. Tillgänglig: https://www.tutorialsteacher.com/core/dotnet-core , Hämtad: maj, 08, 2022

[13] “An introduction to NuGet”, *docs.microsoft.com*, Feb, 02, 2022. [online]. Tillgänglig: https://docs.microsoft.com/en-us/nuget/what-is-nuget , Hämtad: maj, 08, 2022

[14] “Introducing JSON”, *json.org*. [online]. Tillgänglig: https://www.json.org/json-en.html , Hämtad: maj, 12, 2022

[15] G. Dennis, *et al*. “JSON Schema: A Media Type for Describing JSON Documents”, *jsonschema.org*, Jan, 28, 2020. [online]. Tillgänglig: https://json-schema.org/draft/2020-12/json-schemacore.html#rfc.section.4.1 , Hämtad: maj, 13, 2022

[16] “Serialisera och deserialisera (marskalk och ogift) JSON i .NET”, *Microsoft*, Maj, 26, 2022. [online]. Tillgänglig: https://docs.microsoft.com/sv-se/dotnet/standard/serialization/system-text-jsonhow-to?pivots=dotnet-6-0 , Hämtad: maj, 13, 2022

[17] “HTTP Request Methods”, *w3schools*. [online]. Tillgänglig: https://www.w3schools.com/tags/ref_httpmethods.asp , Hämtad: maj, 13, 2022

[18] T. Berners-Lee, *et al*. “Uniform Resource Identifier (URI): Generic Syntax”, *datatracker*, Jan, 2005. [online]. Tillgänglig: https://datatracker.ietf.org/doc/html/rfc3986#section-3 , Hämtad: maj, 16, 2022

[19] “The components of a URL”, *ibm*, Mars, 05, 2021. [online]. Tillgänglig: https://www.ibm.com/docs/en/cics-ts/5.1?topic=concepts-components-url , Hämtad: maj, 16, 2022

[20] “Real-time ASP.NET with SignalR”, *dotnet.microsoft.com*. [online]. Tillgänglig: https://dotnet.microsoft.com/en-us/apps/aspnet/signalr , Hämtad: maj, 20, 2022

[21] “Code faster Work smarter”, *Microsoft*, 2022. [online]. Tillgänglig: https://visualstudio.microsoft.com/vs/ , Hämtad: april, 08, 2022

[22] “Git - local branching on the cheap”, *Git*. [online]. Tillgänglig: https://git-scm.com , Hämtad: april, 11, 2022

[23] IBM, “Docker”, *ibm*, Juni, 23, 2021. [online]. Tillgänglig: https://www.ibm.com/inen/cloud/learn/docker , Hämtad: april, 11, 2022

[24] “Azure DevOps”, *Microsoft*. [online]. Tillgänglig: https://azure.microsoft.com/enus/services/devops/#customer , Hämtad: april, 11, 2022

[25] “Vad betyder agilt arbetssätt (eng. agile)?”, *Stockholm School Of Economics*. [online]. Tillgänglig: https://main.exedsse.se/vad-betyder/agilt-arbetssatt , Hämtad: april, 12, 2022

[26] Cerisegruppen, “Parprogrammering”, *Kungliga Tekniska Högskolan*, Aug, 2020. [online]. Tillgänglig: https://www.csc.kth.se/tcs/projects/cerise/parprogrammering/ , Hämtad: april, 12, 2022

[27] “What is CRUD?”, *sumologic*. [online]. Tillgänglig: https://www.sumologic.com/glossary/crud/ , Hämtad: april, 12, 2022

[28] C. Nienaber, R. Suter, “ASP.NET Core web API documentation with Swagger / OpenAPI”, *Microsoft*, April, 14, 2022. [online]. Tillgänglig: https://docs.microsoft.com/sv-se/aspnet/core/tutorials/web-api-help-pages-using-swagger?view=aspnetcore-6.0 , Hämtad: april, 25, 2022

[29] O. Elgabry, “Database - Design Process (Part 3)”, *Medium*, Sep, 14, 2016. [online]. Tillgänglig: https://medium.com/omarelgabrys-blog/database-design-process-part-3-7b5fafc78774 , Hämtad: maj, 20, 2022

[30] “Vad är ett API?”, *Skatteverket*. [online]. Tillgänglig: https://skatteverket.se/omoss/digitalasamarbeten/utvecklingavapierochoppnadata/kunskapochinspiration/vadarettapi.4.96cca41179bad4b1aaa4b8.html#:~:text=fungerar%20ett%20API-,Ett%20API%20best%C3%A5r%20av%20byggstenar%2C%20s%C3%A5%20kallade%20endpoints%2C%20som%20tillsammans,kan%20dessa%20prata%20med%20varandra , Hämtad: maj, 20, 2022

[31] ReqBin, “HTTP GET Request Method”, *ReqBin*, Juli, 24, 2021. [online]. Tillgänglig: https://reqbin.com/Article/HttpGet#:~:text=GET%20is%20an%20HTTP%20method,on%20data%20on%20the%20server , Hämtad: maj, 21, 2022

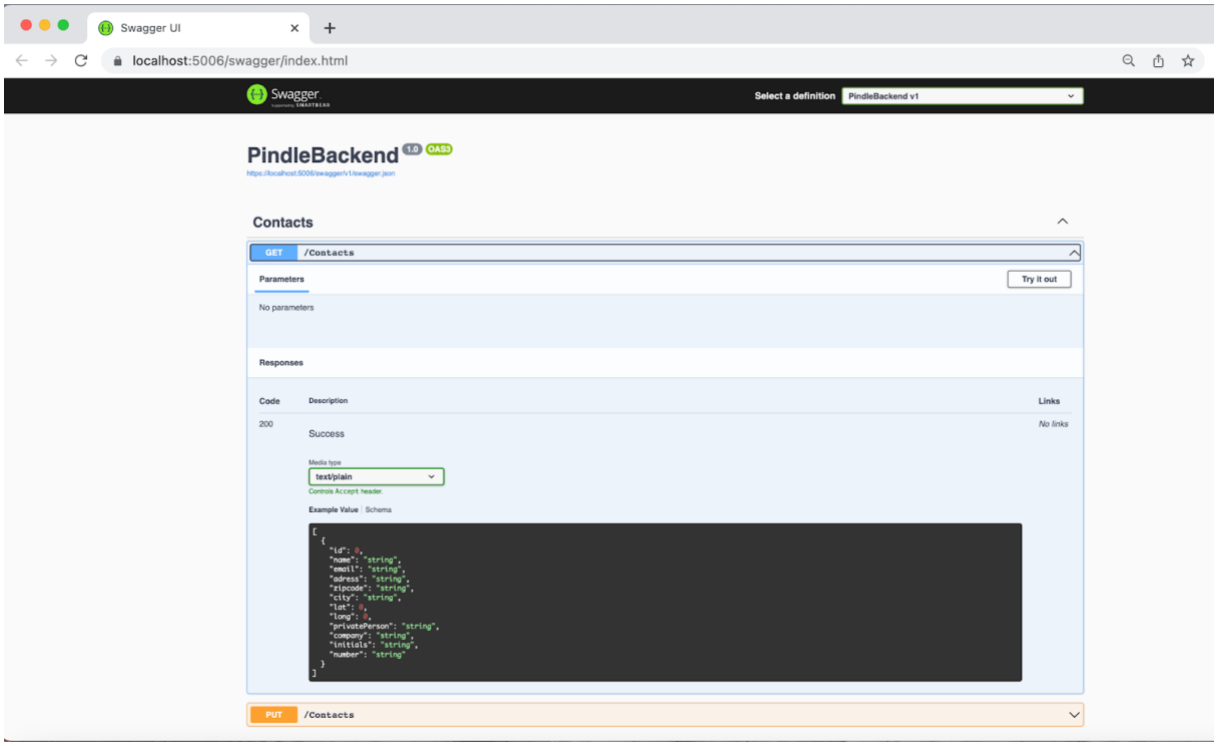
[32] “Put in ASP.NET Core REST API”, *pragimtech*, 2020. [online]. Tillgänglig: https://www.pragimtech.com/blog/blazor/put-in-asp.net-core-rest-api/ , Hämtad: maj, 21, 2022

[33] D. Britch, *el al*. “Display Pop-ups”, *Microsoft*, Aug, 07, 2021. [online]. Tillgänglig: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/user-interface/pop-ups , Hämtad: maj, 24, 2022

[34] “Data Annotations - MaxLength Attribute in EF 6 & EF Core”, *Entity Framework Tutorial*. [online]. Tillgänglig: https://www.entityframeworktutorial.net/code-first/maxlength-minlengthdataannotations-attribute-in-code-first.aspx#:~:text=The%20MaxLength%20attribute%20specifies%20the,%5B%5D%20properties%20of%20an%20entity , Hämtad: maj, 24, 2022

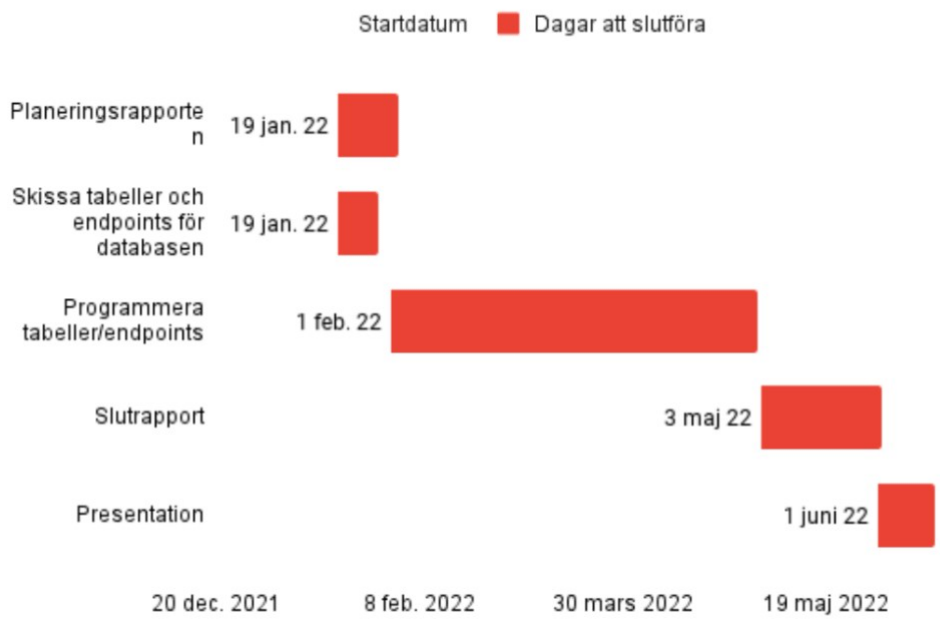
[35] “Online JSON viewer”, *jsonviewer.stack.hu*, [online]. Tillgänglig: http://jsonviewer.stack.hu/ , Hämtad: maj, 24, 2022

Bilagor



Figur 1, en bild på webbsidan Swagger där man kan testa om databasen är igång samt testa sina endpoints

Aktivitet	Startdatum	Dagar att slutföra
Planeringsrapporten	2022-01-19	15
Skissa tabeller och endpoints för databasen	2022-01-19	10
Programmera tabeller/endpoints	2022-02-01	90
Slutrapport	2022-05-03	30
Presentation	2022-06-01	14



Figur 2, en bild på tidsplanen som gjordes innan projektet startade

Länk till tidsplanen:
https://docs.google.com/spreadsheets/d/13hYYI3E5OgNnswLRELSf_saT14Wz43jsTQWfwhPIdCc/edit#gid=0


AVANCEZ
1829
CHALMERS