University College Nordjylland

Technology and Business

Academy Profession Programme in Computer Science

Elective Semester Project

Team: Mobile Dev Group 1
Title: Marnie – Train dating app

Team members:
Oleksandr Karbovskyy
Marius Untaru
Christian Glambæk
Martin Wolter Kaas Nielsen

Supervisor:
Mogens Holm Iversen
MHI@ucn.dk

Hand-in date:
31-05-2017

Abstract:
The report is about the process the group went through to create a dating app for people who use trains. The report has a focus on the learning process the group used during the research and the creation of the app, it will also go into detail on some of the things the group had to focus on when developing the app. Some of these areas are how to use Rejseplanen to get information about trains in Denmark, and how the group could use social media as a login method

Synopsis

This report will go through the development process of a mobile app, that helps people find dates, when traveling by train. The group behind the project will describe the thoughts and problems they went through during the project, with a focus on the learning process, and how they went about limiting the project. They will describe the process they went through when designing the database and its iterations. They will write about how they wanted to implement a login system using a social media network, and the many problems they encountered with this. They will also discuss security in relation to the project and how they wanted to use Rejseplanen to find possible routes for the users, why they went away from that idea again, and what their replacement were. In this project, the group had to choose between three mobile technologies, they will describe the good, the bad, and why they chose what they did. They also did prototypes in the other two mobile technologies, they learned during the semester. The focus of the project was how they go about learning new things, and what the best way to learn for each individual group member.

Table of contents

Problem statement

Nowadays a lot of people are using trains as their main way of transportation, and they spend a lot of time there, sitting alone, playing around with their phone or reading a book and at the same time they are missing out on the opportunity of meeting a new friend or finding their new love.
A convenient solution to this would be a mobile application where people can arrange a dates and meet in the train so they can get the best out of their long time that is spent in the train.

About the app - Marnie

Our app is targeting people who commute by train, that wants to get in contact with other people, who are on the same train as them. We are limiting the app to only work in Denmark, as we wanted this to be our focus area. From our classes, we heard that there are approximately as many Android users as iOS users in Denmark, and for this reason we decided to develop the app as a cross platform app, to reduce the costs and time of developing two native solutions.
The core functionality of the app is going to be the ability to find other people on the route the user has searched for, and then being able to send an invitation to go on a date with the person, the other person will have to do the same thing, for the date to be confirmed. To make it easier for people to find each other on the train, when the date has been confirmed, we had the idea to add chat functionality to the app. We would need a place to keep all this information, which should be available for all users, because of this, we would need a backend database, so the app will have something to pull and send data to.
In general, it is not a good practice to make direct connections between a mobile app and a remote database, because of performance and security reasons. That is why, we decided to make an extra layer, a Web API, between our app and database. This will make the app more lightweight and scalable.

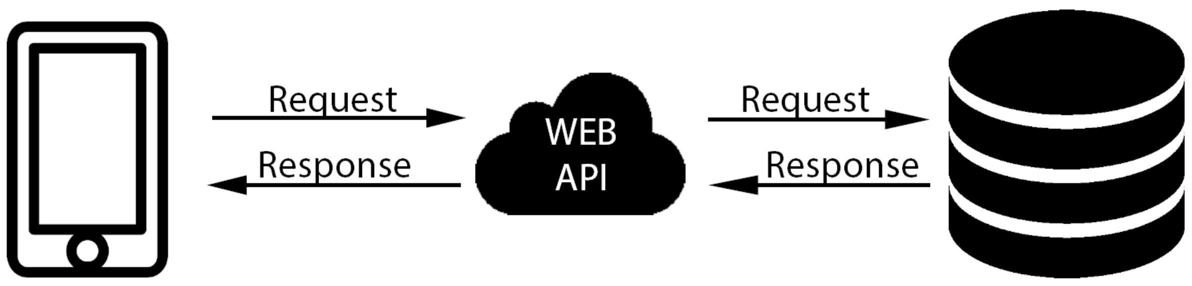


Fig. 1 General architecture of the solution

Learning goals

- How can we use Rejseplanen to find the routes that people are on?
    - We have thought of using the Rejseplanen API. From the research we have made, we can see that it offers a way to find all stations in Denmark, and it can check if there is route between these stations. This way we can get the newest data from Rejseplanen, since it's owned by all public transportation companies in Denmark.
- How can we create a login that uses Facebook or Google to make the signup process easier?
    - From our research, it shows that we could use OAuth, Firebase, Native login or create our own login system.
- How can we use the GPS to find the nearest station to our location?
    - We can see that Rejseplanen offers a service to this in their API, so we are going to try using it, an alternative would be to build our own database with all train stations in Denmark, and match the phone's location to that.
    - We can use Rejseplanen API and get all stations within a given radius from the current location
    - How to get current device location
- How can we create a private chat between two people that are connected with each other?
    - Our research shows that there are a lot of different ways to make a chat. What we have considered is three completely different ways of creating a way to chat.
    - SignalR which is a ASP.Net library to make real time communication. This would work well together with the Xamarin approach.
    - Firebase which also seems to be a way to create a chat, this could work together with the firebase login.
    - A third possibility would be to create our own chat server.
- How to sync local mobile database with the remote database?

    - Our research shows there are at least two different approaches to this. First one is to write the synchronization ourselves and the second one is to use third party tools.
- Which technology should we use to create the app?
    - We learned 3 technologies for developing apps: Xamarin, React-native and Android. We will research which is best suited for this app, do prototypes in each to find out what works best and discuss which one to use.

Limitation of project

After having decided what features were the must have in the Marnie app, we began to research, what it would take to implement these features, and we started to estimate how long time it would take versus how much time we had. As mentioned before the core feature of the app will be sending a date invitation to another person who is on the same train, at the same time as you. This involves making a train/route search function, and we considered this to be the must have feature of our app, as the app would be nonfunctional without them. Another important part of the app is user identification, this involves a login and registration system, besides this we need some basic personal information to be registered as well, because a user needs some basic information to form an opinion on whether or not to go on a date with another user. These features are considered as important.
As almost all functionality of our app relies on the backend we decided to start with implementing the database and our Web API.
Our research on how to create a chat component for the app shows that we would need some kind of real time database, and that the implementation of the chat wouldn't be as easy as we thought. As it was not one of the must have features and since none of the group members had any experience working with it before, we decided to add the chat to our nice to have feature list, as it would be nice to have in the app. We also decided that most of the UI should be made after we have handed in the project, as it could take a long time, and it wasn't important for the report that it was styled. This means that some of the UI elements are very basic.

Choosing the development technology for the app

In this section, we compare our experience that we got from the semester and chose what technologies we will use to create our main app.

Our experience from our courses and workshops

During the fourth semester, we participated in three courses and one workshop about mobile development. From this we gained a lot of knowledge about mobile app development, and using different approaches in the way you choose your platform and development technology.

Native Android

The first course we were participating in was Native Android development, using Android Studio which is created by Google and Jetbrains. Android Studio is the official development platform for creating Android apps, that uses Java as it main programming language. But even though it's using Java, it is very different creating an Android app compared to the programs we made in Java on our first two semesters, but knowing Java made it easier to get started. During the course, we learned the basic of how an Android app works, like what happens when the user interacts with the phone or when the user wants to use the camera. We also learned how to use a local database, and how to handle internet resources. These small things that we learned, was nice to know, but it wasn't enough to make a marketable app. We had to go out and do a lot of research by reading the official documentation and by finding guides, we found the documentation on the official site to be excellent, and they often also provided a guide on how to do the things we looked up. We also had a lot of time with no classes, where we could do some of our own research on things we wanted to learn. During this course, we also got an assignment, where we had to develop an app for a macro computer called the Mono. The Mono had some sensors that could measure the temperature, humidity and acceleration, and we had to create an app that could access these parameters and display them in a graph.

Xamarin

The second course we had was about Xamarin, and how to make a cross platform mobile apps using .Net technologies and C# as the programming language. We felt comfortable with starting to learn Xamarin, as we had a good basis in C# and .Net from the previous semester. The way the user interface is built in Xamarin resembles how it is built in Native Android, and how the interface of a WPF program is made, so it was easy for us to pick up on. We also found that the Xamarin community is very mature, with a lot of guides and components, for a lot of the things we needed. During this course, we also learn how to work with the things we learned in the Android course, but using Xamarin this time. We also got to work on a app for a plumbing company, that should make it easier for a worker to register their installations of new water meters. At the end of the course we had a successful presentation of our prototype to the customer, they were very interested in further development of the app.

React Native

We also participated in a workshop where the goal was to get familiar with React Native, which is based on JavaScript, and it was very different from the other languages we have learned like C# and Java, because these languages are Object-oriented. Before we had this workshop, we were a part of the WebMobile1 course, where we got an introduction to HTML, CSS and JavaScript, but there wasn't enough about JavaScript to start coding in React Native. React Native is a fairly new cross platform technology, that is still under heavy development. It is missing a lot of features that Xamarin and Native Android provide.

We got an assignment as well, which was to make a quiz app. We managed to develop a very basic app at the end of the workshop having spent a lot of time and energy on it. We describe our experience with React Native in more details under Prototyping.

Choosing the main development technology

We decided that the app should be a cross platform app, as it made the most sense that a dating app is available on the most platforms, so because of this we had to choose between Xamarin and React Native. From the experience, we had from our courses, we felt that Xamarin would be the best choice for us, as it was the one we felt the most comfortable with. The platform felt more mature by having a lot of guides and better documentation, as compared to React Native, that is still under heavy development. There isn’t a lot of React Native guides and the documentation is very basic. Performance wise they both seemed to be around the same, though React Native is known to have a shorter compile time and the possibility of seeing the effects of any changes made to the code, instantly.

Building the backend and Web API

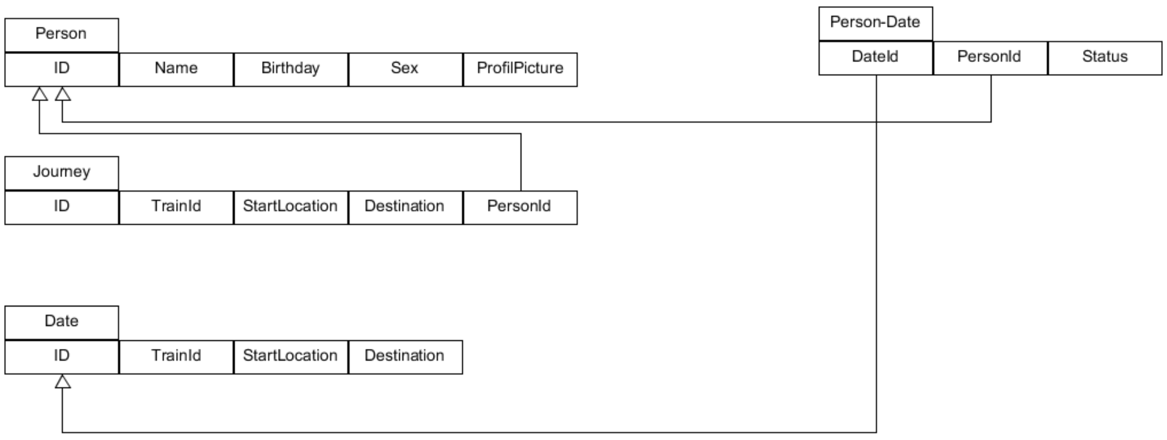
We wanted most of the functionality of the program to be on the backend, so the solution was easier to build. This gives us the flexibility to add any type of client, for example a website, if it was needed later to reach more customers. If we found out during the process, that another language was better to build the API in, it would have been easy to switch to the other language as well, or move to another database without influencing the app.
We decided that the best way to build this app, would be to use an API to make all the calls to the database, and not just sending raw SQL queries from the phone. We also figured out that by having most of the functionality on the API, that it would minimize the use of the battery on the phone, which was something that was important for us.
The API is built using C# and it’s using JSON as the format it sends and receives in. The API is hosted on an external server.
We found out early in the process that we would need to run the API on an external server, as you couldn’t communicate with the API when it was on a local machine, using a real device. We also found this to be a good idea, as it was how the release version should run.
The use of API’s was not a new topic for the group as we had some experience from last semester where we learned, in a workshop, to work with restful API’s. We used some tutorials from internet[Ref 2] to get the idea how to build a Web API to communicate with the database in .Net framework. Beside that creating and using a Web API is very similar to a WCF service, which we learned and got experience with last semester.

Database

An overview of our process of creating the database is presented below, how we did it and what choices we made.

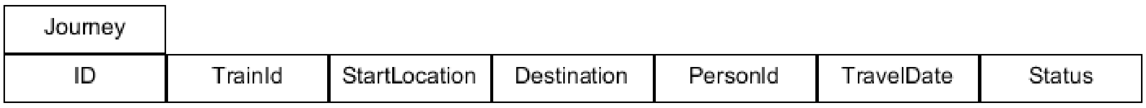
Using Entity Framework

As we were new to Entity Framework we decided to try to use a code first approach to generate a local database to test with. After learning how to do that, and succeeding, we uploaded the database to a hosting service. After hosting the database any future change was done manually. When we implemented our own version of Rejseplanen the database changed a lot, we updated our code to match the new database schema. To verify the relations between model classes and database, we made a new project using the database first approach, and compared it to our current code.



Database diagram v 1.0 for train dating.

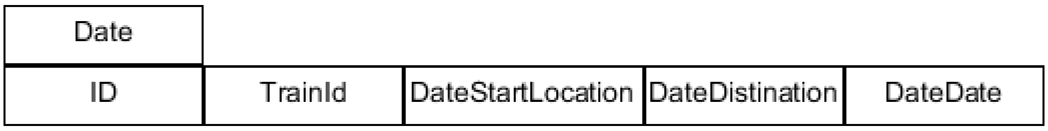
Our first version of the database was based on using the Rejseplanen API to get the travel information. What we needed was to save some basic information about the person like name, birthday, gender and a profile picture. We also needed to save the journey that the user is going on, and what stations the train departs and arrives at. This would give us the option to be able to compare two user's journeys, and see if they could go on a date. The last thing we needed was to save the Date, which contains a Train ID, start location, and destination. There is also a Person-Date table in our database, this is used to hold the person's date and to hold the status of the date, to see if it was confirmed or not. The reason for having a start location and destination on both the Journey and the Date is that when we combined the two user's journeys into one date, then we could get the start location from one of the user's journey, and the destination from the other user. This is the reason why the date needed its own separate locations.



Journey						
ID	TrainId	StartLocation	Destination	PersonId	TravelDate	Status

Changes to Journey from Database diagram v 1.1 for train dating.

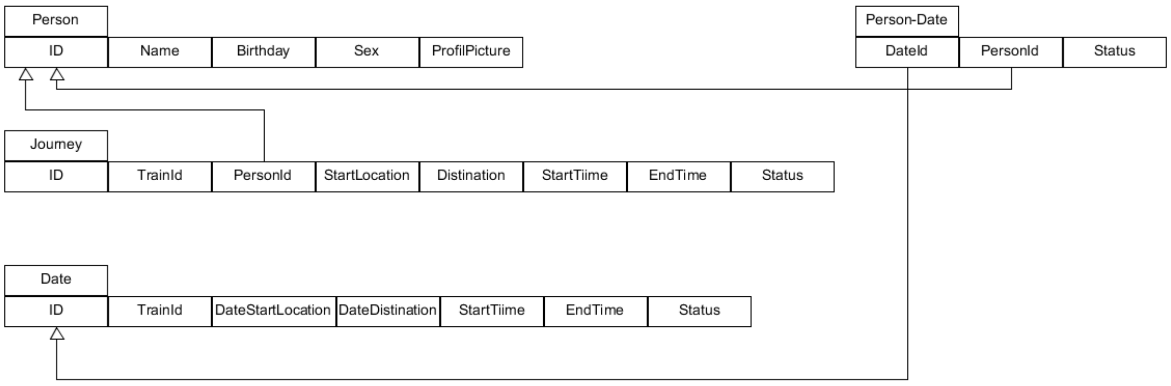
We found out by looking through the DSB and Rejseplanen documents that the trains have the same id every day for the same route, so we figured out that we needed to add a TravelDate field to the database, that contained the date and the departure time. The status was also added, this would be a way of showing if you were still interested in finding new dates on your journey.



Date				
ID	TrainId	DateStartLocation	DateDistination	DateDate

Changes to Journey from Database diagram v 1.2 for train dating.

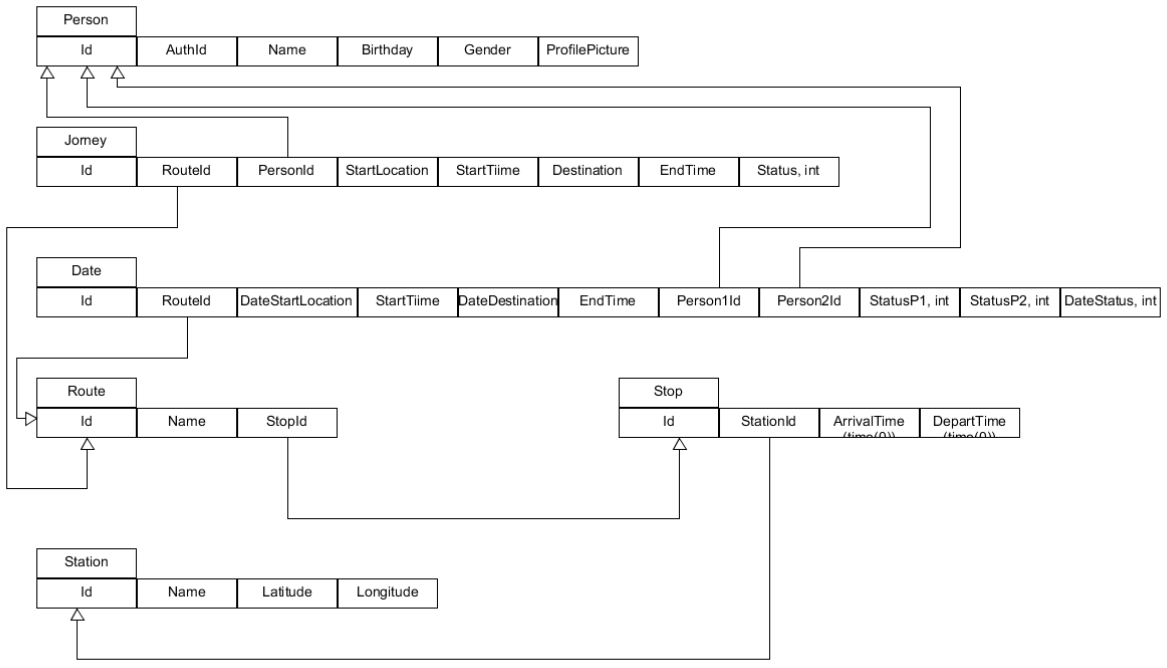
Our Date table also needed a date and time, so we could display the time to the users, so they could plan their meetings on the train, as the date is not attached to the journey.



Database diagram v 2.0 for train dating.

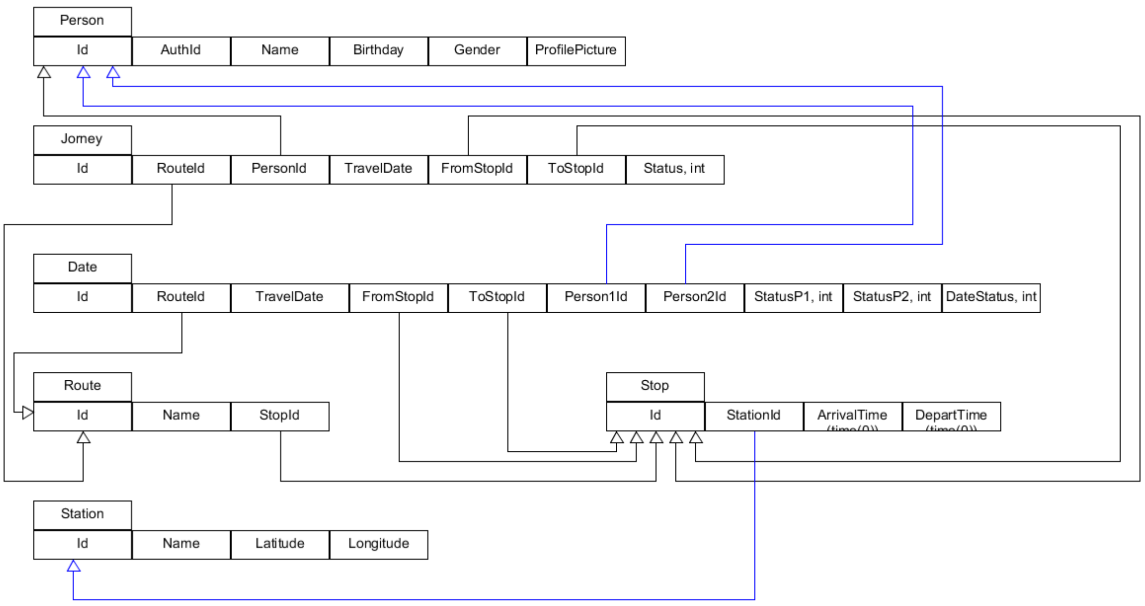
In the second version, we realized we could use DateTime and TrainId to compare two users Journeys and see if they could go on a date. This meant that we removed TravelDate from Journey and replaced it with a start time and end time. We also added this to the Date, as it would indicate to the user at what time they could meet with the other person. A thing we could have done to optimize the database at this point, was to add a new table with Location and Time, and make the Journey reference this new table.

We encountered a problem at this point of the database, with the JsonSerializer, where it would turn the date and time it got into UTC. We tried to fix this on the API side, by define the Danish time zone, but there still was a two-hour time difference between Danish time on the client and UTC on the API. The way we fixed this was by keeping all DateTimes in UTC.



Database diagram current version.

This is our current working version of our database. In the end, we decided that only two users can go on a date together, so we change the many to many relations between Date and Person, to a many to one relation. This lead us to change the Date table, which now have two person IDs for each date, and a status for each person. It was also at this point that we decided that we wouldn't wait for Rejseplanen to approve us any longer, so we had to add some new tables, Route, Station and Stop. We decided to follow the way DSB does their train IDs, where the route is the ID, and not the specific train, this way we can also reuse the route for every day, since the route is the at the same time every day. We did a lot of discussing before we came to this version of the database.



Database diagram optimized

The current version of the database is not as optimized as it should be. Right now, we use strings in all of our locations in both Journey and Date. What we should have done, was using the Station table that we already have, and use the stations IDs as references in the location columns. We should have done this change, when we decided to drop Rejseplanen, as it would be the best place in our database timeline to have made the switch. We only thought about this change when we were too far into the project.

Web API security

It is a common for a mobile app to use API calls to communicate with backend services and databases. When creating a mobile application which uses a Web API, the security aspects should be taken into consideration. Security in itself is a very huge, complicated, tricky topic and can be a subject for a whole separate project. We don't implement any kind of security in our Web API due to time limitation. But we have read some articles about Web API security and made some considerations about what could be done to protect our webservice and database from unauthorized access.

Using HTTPS

We did some research on how to implement HTTPS on the API, and what we found is that it is easy to set up, all we would need is a domain that we have control over. For this project, we found a free host called MyASP.net where we decided to host both our API and MSSQL database. This hosting service provides only regular unsecure HTTP connection, but we found it fine enough for our project since most of the data is just fabricated, and not a real user's information. But that doesn't mean that we shouldn't worry about encryption of the user's data at all, and try to hide it from people who are trying to sniff out random information. If

we would have to release this app, we would move the things to another service that allows us to use HTTPS.
HTTPS is the secure HTTP protocol which allows clients to send data over the network in an encrypted state. We will not explain how HTTPS protocol works and how data is being encrypted/decrypted as it is outside our subject, but will say a few words about why it is useful in our case. Unlike HTTP where data is being sent as a plain text, HTTPS protects against “man in the middle attacks”, when the enemy can sniff the data while being sent over the network, you avoid this by encrypting the data using special encryption algorithms.
Although it doesn’t protect against cross-site-request-forgery. CSRF is an attack which tricks the user (mostly using social engineering) to perform malicious actions against a web service or database. As our research shows, CSRF is applicable only for web apps or hybrid browser based apps which use cookies. Despite CSRF is not quite relevant to our app there is still a common practice to implement authentication and authorization mechanisms with tokens when accessing Web API.

Authentication and authorization

Authentication is the mechanism which allows to verify the user, typically by using a username and password [Ref 7]. If a user has been verified successfully the authorization process takes place by sending a token to the user as a response. The token can be a special encrypted key which only the service can understand. When the user makes an API call he includes the provided token into request header. The service then verifies the token and, if it is successful, it authorizes the user.
The token is stored on the server side for future use. It usually has an expiration time, and when it expires the user must be authenticated to the system again.
Similar scenario concerns to the login part of our app when connecting to the Auth0 service. But in addition to token, a unique client ID is being added to every authentication or authorization API request. Client ID is provided by Auth0 when registering an app. This adds an extra security layer by verifying not only the user but the app as well.
The described scenario could be implemented in our Web API. There are two ways of doing this. First one is to implement our own login system on the API side, another one is to keep Auth0’s user verification and implement only authorization part to our API. In the second case the smartest way to pass a token from Auth0 to our Web API, hash it and save it into the database should be implemented. But saving of static tokens can also be a huge security issue. There are some techniques which allow to fix the mentioned issue described in the article “Mobile API security techniques” [Ref 7].

SQL Injections

Because of our use of a database in this project, we must also think about protecting us against SQL injections [Ref 8]. SQL injections is one of the most common hacking techniques used against SQL databases. As an example of SQL injections, one person can place a malicious SQL

statement into an input field, and affect or even destroy a database. What we have done to protect ourselves against SQL injections is by using parameterized SQL statements. What this does is it makes sure that all parameters that are being passed from a user’s input to an SQL statement, are always properly escaped. Another thing we could have done was, verifying the user's input, this means that we could ban some illegal characters, like semi colons and asterisks.
We are using middleware, Entity Framework, between our Web API and our database. Using Entity Framework, gives us the opportunity to use LINQ statements instead of SQL queries to communicate with the database. Entity Framework converts the LINQ statements into parameterized SQL statements for us automatically behind the scene.
We didn’t have to go out and find any new information about prevention of SQL injections, as some of the group members work with this a lot last semester.

Implementation of signup and login

For the authentication method we considered using Firebase, Auth0 or creating our own method. We also thought about using social media platforms as Facebook or Google+

Firebase

Firebase is a cloud platform created and supported by Google, which provides a wide abundance of services including real time databases, authentication services, cloud messaging and much more. Firebase provides its own SDK for Android and iOS.
The .NET platforms are not supported, so it is not compatible with Xamarin. Firebase also provides a Rest API, this could have been an option for us, but after some more research we found out that Firebase API does not provide a possibility to create a new user. This was a very big disadvantage in our case.

Our own login system

Creating our own login system would require that we implemented an encryption algorithm, and we would have to get a secure connection to our webservice and database, to be able to create, send and save a user's authentication credentials securely. As none of the group members had any experience with this, we decided not to go this way, as the time spend on this could be too much, compared to some of the more important features of the project.

Facebook

There are two ways to implement a Facebook login system into an app.
The first is to use their own SDK that they provide, but unfortunately, this approach was not possible with Xamarin as the SDK only supports native platforms, so we would have to mix our Xamarin code with native Android code. Another problem is that the Facebook SDK requires the user to have the Facebook or Facebook Lite app install on their devices, as it uses the app

as the login method, we didn't want this, because a lot of users don't have the Facebook app installed and some people, like Christian, who has a third-party Facebook app, and this means he wouldn't be able to login.
The second way would be to build our own login flow using the Facebooks API, due to our app being cross platform, we decided to try this approach. We went out and found a video from MSDN Channel 9[Ref 1], where they showed how to do it, and we also went and read the official documentation on Facebook's developer site. There was also some official Xamarin tutorials that we tried to use.
Our original plan was to add a button that would either open the Facebook app if it was installed, or it would open a WebView, and use that to identify the user for the login system. We would also had used Facebook as a way to create an account, because we thought we could get all the basic information that we need about a new user. Sadly, if we wanted to get the user's birthday or gender, then we would have had to get our app approved by Facebook. This process seemed like it would take a lot of time, and there was no guarantee that we would get approved, and since the login was a very important part of the app, we decided that we would try something else.

One thing we could have done, to prevent getting approved by Facebook, would have been to use Facebook, to get the user's name, age range and profile picture, and then get the user to enter the rest of the information like birthday and gender, then send all this information to our database. We thought about using this, but in the end, we went away from this again, as it seemed to be too much of a hassle for the user, having to first authenticate with Facebook, and then enter a bunch of information, instead of just using one registration form.

Other social medias

After we tried using Facebook as a login method, we tried to do some research on other social media that we use for logins, that had the same information as Facebook would, and where we could access this information without having to go through an approval process. The closest one to Facebook we found is Google+. Google+ offers a lot of the same information as Facebook does, and they also have a lot of users, i.e. people with Google accounts. The biggest problem we found with using Google+, is that all the information that we wanted, needed to be entered by the users into their Google+ account, before they signed up and since nobody really uses Google+, that also means that nobody has filled out their account on Google+. So, in the end we decided that it wasn't a good idea to implement a social media network into our app during this project. If we had more time to be approved, then it might have been more feasible option.

Auth0

While we were doing research on methods to use social media networks as logins, we came across a site called Auth0. We chose to use Auth0 because they provide a secure way to create

and save login credentials, authenticating users and provides a service to use social media sites as login, by going through the Auth0 site. We tried to use them for our Facebook login, but as mentioned earlier, we dropped that. Auth0 documentation [Ref 5] on how to use their API was very helpful and it made it seem easier to implement their API into our app.
We used their service where they provide a login functionality that needs an e-mail and a password for the login, if the user logs in correctly, then a token is returned, we use the token to send it back to the API, to request a UserId. We use the UserId in our database to link the login and person. The UserId is a unique string of 24 letters and numbers.

Saving log in status

We decided that we would save the user's login status once they logged in, so the user wouldn't have to login, every time they open the app. The way we tackled this was by using application properties, and saving the login status as a Boolean variable. On application start the app checks for the state of the Boolean, if it is true, then the app navigates to the start screen, if it is false, then the login screen will be presented to the user.
Application properties is the local persistent storage build into Xamarin Forms, and it makes the properties available everywhere. It works like a regular dictionary data structure, and it makes it possible to save data as a key value pair. It gives the app an opportunity to save primitive data types and objects in JSON. We learned how to use it by following a guide from Code Mill Technologies[Ref 4].

Travel search implementation

An important part of this project was to have the ability to search for a train, using Rejseplanen, to make sure that the trip was possible.

Rejseplanen

In our problem statement, it says that we would like to use Rejseplanen API, because it gave us the possibilities of finding all stations in Denmark, and we could use it to check if the route, the user inputs, is possible. We found out after some researching, that we would need to be approved by Rejseplanen to be able to use their API, so we went through their registration form and got a message saying it could take up to a week for an answer, we waited for a few days, and decided that it would be better for us to make our own API that offered the same features that we would need from Rejseplanen.

Building our own version of Rejseplanen

The way we tackled making our own version of Rejseplanen, was by making a small-scale database, that have seven stations, and with four possible routes. We decided to make our version close to the way that Rejseplanen handles their API, this way we could continue to make the app like we have envisioned it from the start. Making our own version also meant that we would have to spend extra time developing the API, because we had to figure out how Rejseplanen does the things we wanted from them.

Finding the nearest station

In our problem statement, we stated that we wanted to find the nearest station to the user, for when they have to find a train. We wrote that we wanted to use Rejseplanen for this, since they offered this in their app. We dug through their API and found out that it seemed like they offered this service, but they didn’t, and since it took a lot of time to get approved to use their API, we decide that we would make this for our own API.

We discussed what would be the best way to do this for our project, and what would be the best if we had to take the project to a full-scale app. In the end, we came up with two ways to do this, that both have some good and bad sides.

The first way would be to use a local database that had the coordinates and name of all stations, and each time the app was opened, it would look through the database and find the nearest station to the user. This would be fast since the database would be on the phone, but the bigger the database got, the slower the search would be, and it could use a lot of the battery. By using a local database, it means that the search works offline, and this save us a call to the API.

The second way would be to call the API and make that do all the filtering. This could take longer depending on how fast the phone's connection to the mobile network is, but by using the API for this we save space on how big the app would be, and we can update the database and search algorithm without having to update the app.

In the end, we ended up going with the second way, since the local database would use too much power, because of all calculation it would have to make. By using our API, we save power, and we only get one station returned, and not a list of all the stations.

We have been studying how DSB finds the nearest station in their app, this where we got the idea from, and they have a local database on the phone, we think they chose this because it’s fast and because of the way they have set up the app, they can get the nearest station before the user gets to the search screen, making it seem like it’s fast.

A way we could make the search faster and save battery while using a local database, would be to make a database call that only returns the list of stations within a given radius, of the phones coordinates. So, the algorithm has fewer stations to sort through. Instead of comparing with all stations in Denmark, which is what the API does right now.

We learned during our Xamarin course how to use the GPS in Xamarin. This is done using a NuGet package called Geolocator[Ref 3], and we used their guide to setup the precision of the GPS. In our case, we don't need the most precise position, because of the distance between train stations is often very big. We also had to make a method that could sort all the stations to what was closest to your coordinates, we did a bit of research and found two ways to do it, one was to use a difficult math algorithm, and the other option was to use GeoCoordinates [Ref 3], which is a class that is available from the .Net Framework 4.0. We decided to use the last one, as we found an example on Stack Overflow. We also looked into the documentation MSDN, to get some more knowledge on how the class worked.

Finding stations in the search

A nice to have feature we have been thinking about that could make the train search better, would be that when the user inputs the station name into one of the fields, that there would be a small dropdown that starts showing stations that are similar to what the user has inputted. This would be implemented by making a call to the API for the TrainSearch screen where station input is required and get a list of possible stations. We should add a small countdown timer, so the app doesn't filter the list every time the user inputs a new character. We could also use a local database for this, which would make more sense if we also use that database to find the nearest station.

Creating the date system

When creating the dating system, there were two focuses, one was on creating an ethical system that didn't make people feel rejected. The other was on making sure that people could meet on the train.

How to make the best dating invitation system

We have discussed two ways to make our date invitation system.
The first way would be that when the first person presses the date button, the date is saved in the database and the second person gets a notification that the there is a date invite. If the person accepts the invite, the client would send an UPDATE request to the API and the date would be updated, and both would get a notification saying that the date is confirmed. The problem with this one, is that the person sending the date invitation knows that the other must say yes or no, and if the person said no, the first person might feel rejected. The person could confront you on the train, asking them why you said no, or something worse.

The second way would be that both persons would have to say yes, to each other, without them ever getting a notification, before they both say yes. From the client side this would

work a bit different. When the first person says yes, a POST request is sent to the API, then the API has to check: if this is a new date or if it already exists in the database. This is done by looking up a date between these two people on this route. If one exists, the API updates it, otherwise a new date is inserted. When the second person says yes, the client posts to the API and the same check is applied, which leads to the date being updated and both persons getting a notification.
There is something more ethical to this method. If the second person never gets a notification, about the first person wanting to go on a date, then the first person would never know if the second person saw the list of possible dates with the first person on it, and this way no one is rejected, they are just not picked.
We ended up going with the second one because we considered that is the best option from an ethical perspective.

Making sure that people can't date if they are not on the train at the same time

We figured out that each train should have an ID, and we decided to follow the way DSB does their IDs, where a train has the same ID every day, on the time and route. A problem we encountered is that if people are going with train 1 from Aalborg to Hobro, and another person from Hobro to Randers, then the train would have the same ID, and the persons would be allowed to meet, even though they couldn't. Our first thought was to use the Stations to figure out if two persons could meet. After trying to figure out how to make a station grid, we came to the simple realization, that the departure time and arrival time was easier to compare. So, by using the Train Id, departure time and arrival time, we could compare the times to see if people would be on the same train at the same time.

Filled database with dates

If you ask a person on a date, a new object will be created in the database, if the person doesn't answer, then the object would stay there. We have discussed it and the optimal way to handle this would be that a program would check if a date isn't confirmed after the date should have ended, and it would then remove the object from the database. This way we can keep the people who actually went on a date in the database, and all the dates that didn't happen would be removed.

Localization

We decided to make the app in two languages, since we are a multi-language group and this would also increase the usefulness of the app, as international students or tourists would be able to use the app in a real case scenario. This is done using the globalization and localization concepts in Xamarin. By applying the globalization concept, we were able to adapt the displayed text of the app in relation to the language of the device that the app runs on. If the default language of the device would be English, then the text within the app would be displayed in English and the same goes for any other language. By doing so, we were able to use the advantage of not having any hard-coded text anywhere in the app, occupying the RAM memory of the device. Instead, all the text would be retrieved from separate files containing sets of strings already translated into multiple languages. The two languages that we have are Danish and English, stored in separate RESX resource files as part of the localization process that deals with binding the resource files to the layout.

Based on the official guide [Ref 14] from Xamarin, the way the localization concept was implemented and the way it works is by adding at first the default resource files. These are the RESX file that would hold the set of strings specific to the default language of the app and an automatically generated class that holds references to help binding with the XML layout files. For every other language that the app should support, another set of RESX files is added, containing the translated version of the same strings from the default RESX file. The RESX files were edited using the built-in RESX editor from Visual Studio, in absence of this editor the files should've been written in an XML format, where every string is written as an XML data element.
The next step was to identify the language of the device that the app runs on. This is possible only with some platform-specific code because of the different ways every platform can identify the language of the device. The class named Localize was added to the Android project as a DependencyService where with the help of Java.Util.Locale.Default the device locale is exposed. Afterwards the locale is changed to a .NET accepted locale through the AndroidToDotnetLanguage method to avoid any problems in case the locale of the device is valid in Android, but not valid in .NET. This class is later accessed through the ILocalize interface in the PCL project.
TranslateExtension is the class that exposes the RESX resources to XAML by using a resource manager to provide the proper resources for the layout with the help of the DependencyService and the ILocalize interface.

From our research of the localization process we found out that, because of the different gender variations in other languages a good practice would be to assign numbers to genders instead of using strings. As in English: Male, Female and Other would become 0, 1 and 2.
Since Marnie supports only English and Danish we considered that there is no need to go into greater detail with this and because we realized about it late in the development of the app and it would take some time to change the database and methods to make sure that the translations are working. What we could have done would have been to use the ISO 5128 standard to identify the gender based on a int, where a male is 1, a female is 2, 0 is unknown and 9 is applicable. (ref)

How to handle profile pictures

In our database, there is a column for profile pictures, and we have been discussion how we should handle these pictures.

- We could convert the pictures to be byte arrays, and save them directly in the database.
- We could make an upload process that saves the picture on the same server as the API is hosted.
- We could use a third-party image hosting service, like Imgur

We decided that the profile picture would be a nice to have feature, so it would be one that we would implement if we had time, but we think that the last-mentioned solution, where we would use Imgur as a host, would be the most fitting for the project. Some of the group's members have worked with their API before, in another semester project, and know how to handle it.

Prototyping

For this project, we had to make three prototypes, one in each of the languages we have learned during this semester. We were pretty confident in making the real app in Xamarin/C# so that's what we started with, and we finished the app before prototyping the other languages. We didn’t feel that comfortable with React Native, we didn’t have as much experience with it, or JavaScript. As mentioned earlier we didn’t want to make the full app in native Android, since we had decided that the app should be cross platform.

Native Android

After the Xamarin app was done we started working on the native Android prototype. The first problem we encountered was how to login, we found out that we could use Auth0, the same tool we used for Xamarin. With the help of Auth0, implementing a login form was not an issue. We were able to connect the native Android app as a new client to the Auth0 API and create a LoginActivity where a login method would have the email and password of the user as parameters extracted from two simple text fields and then use the auth0_client_id and the auth0_domain to connect to the API. The code samples from Auth0 were very useful and made the implementation very easy and effective. [Ref 9]

We built the train search page, and found that we missed the Xamarin way of sending data to the next page as objects or parameters in a constructor. So, we decided to parse the search criteria to the TrainFound page, which allowed us to call the API, to get a route list, from the activity where we needed to display the route list. At first, we put a stringArray on the intend to parse the search criteria, but after using Gson to parse strings from the API into objects, we found that we could use that here as well. We used the Volley library to setup the call to the

API, and Gson library to serialize and deserialize data. This worked much like the Newtonsoft.Json and Restsharp packages for Xamarin.

The Volley networking library

Since we are talking about developing a mobile app where battery life and many other factors have a huge impact on the user experience, we wanted to use the best networking library in order to avoid wasting resources or getting any unexpected errors and having an overall bad connection between the app and the API. From our research on the web, we found out that the Volley library would be the best performing networking library in comparison with other alternatives like the Apache: org.apache.http.client library or the Java class java.net.HttpURLConnection. [Ref 10]
Again, a lot of documentation was available on the web and implementing GET and POST request was not an issue. The issue came later when we wanted to convert Java objects into the JSON format.

Using the Gson library

To solve the issue mentioned above, we found out a way of parsing JSON by using the Gson library. And again, for performance reasons, we chose Gson for parsing the objects since our app deals with small requests and Gson would handle the fastest parsing compared to other alternatives like Jackson, which is faster when it handles bigger JSON files. [Ref 11]

Using a Custom ArrayAdapter

In order to display a list with all the routes that the user searched for, a new ListView Activity has been created. Here, we faced another challenge as every row of the list had to match our data model and display the right data.
For this we made a custom array adapter that would adapt the data model to every row of the list. This allowed us to have an XML based layout file where we could have multiple views inside a single row of the list. The views would be found using the findViewById() and filled with data by the getView() method inside the activity's implementation.
An article from Vogella[Ref 12] helped us better understand how this would work and how to implement it.

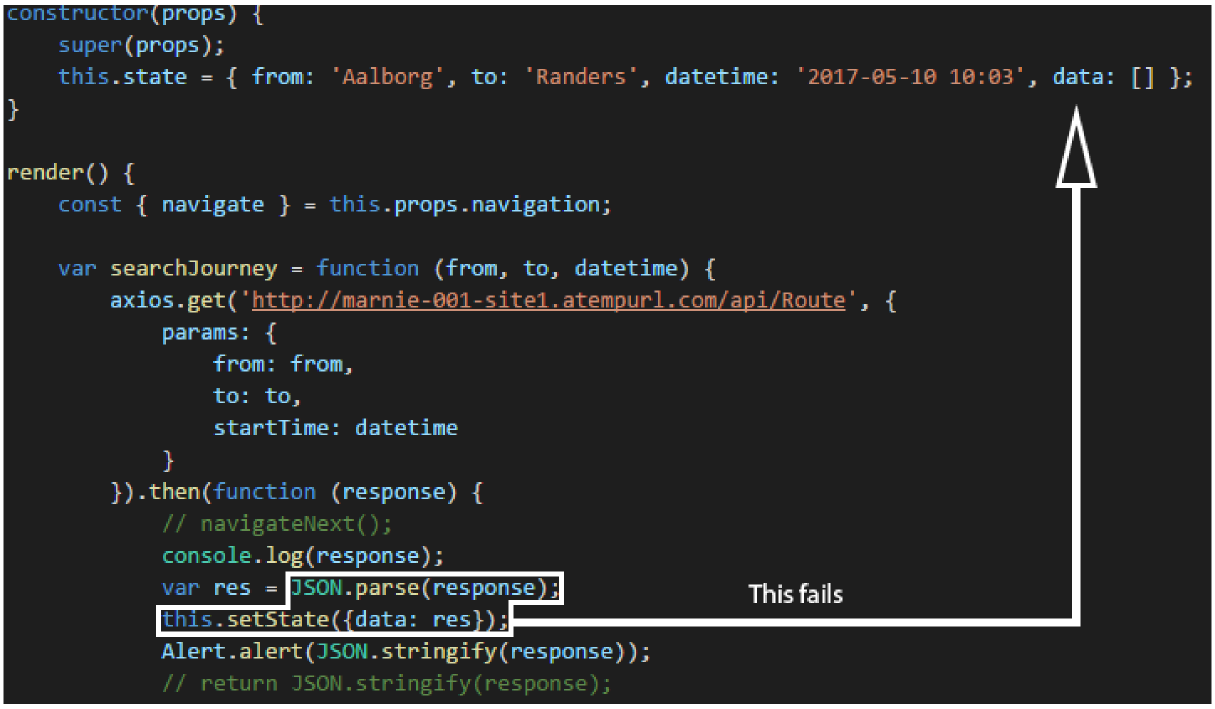
React Native

Our relationship with React Native, isn’t the best, we don’t feel comfortable when writing apps in it, and from trying to make the prototype it has not gotten any better.
During the workshop, we managed to make a very basic quiz app with underlying local database. But the knowledge and experience we got was too poor to manage to make even a deserving prototype of our app. But anyway, we made an attempt and want to share our experience further.

How you build the UI is very similar to how you do it in Android and Xamarin, who both use their own version of XML to present the UI. There is a great function in React, that allows the code to be hot reloaded without compiling the code again, and it works instantly.

While we were making our prototype in React Native, we ran into a lot of problems, and these were hard to debug, because when we activated the build in debugging tools, the app became more unstable, and would be more prone to crash at random times. We tried to use the internet to solve our problems, but since React Native is still a new framework, the documentation for errors are still hard to find, and some of the answers don't work anymore, because the framework has been updated, and now it works in another way.

We have implemented two screens in our prototype. The login screen and the train search function. The login screen is very simple, it has two text fields for the user's login credentials, and a login button. The button calls the Auth0 API.
The train search screen consists of three text fields, one for the origin station, one for the destination station, and the last is the time plus date. This field is a normal field that gets sent to the API, we didn't implement a time or date picker here, since it required us to install a library for it to work on both iOS and Android. When it came to handling the result, we got back from this screen we ran into multiple problems.
The first problem we encountered was that we couldn't pass our JSON response we got from our API to the list view we tried to create. We couldn't do this because the response from the API isn't visible to the UI, and we couldn't assign it to the properties either. In the code snippet below the JSON.parse command fails with an unexpected end of JSON body. The "this.setState "command should assign the response to the data property, but it does not work either as the data array is still empty after this.

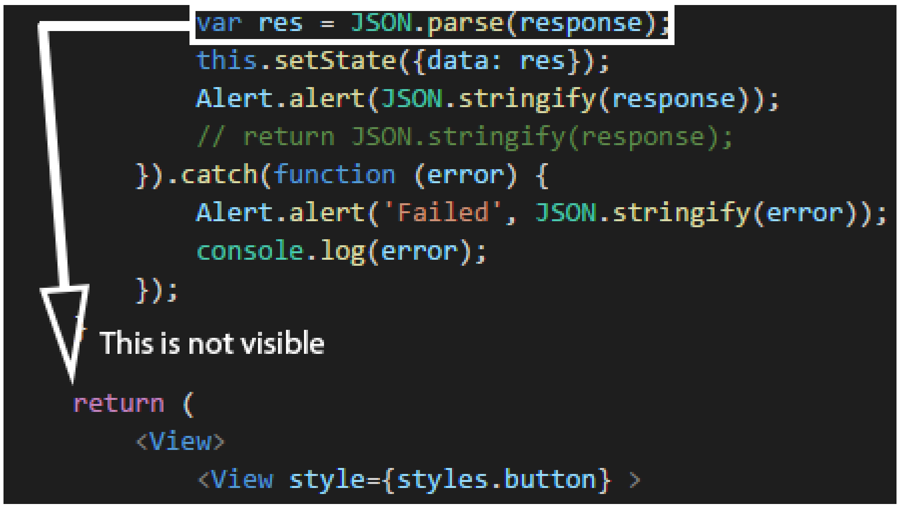


```
constructor(props) {
    super(props);
    this.state = { from: 'Aalborg', to: 'Randers', datetime: '2017-05-10 10:03', data: [] };
}

render() {
    const { navigate } = this.props.navigation;

    var searchJourney = function (from, to, datetime) {
        axios.get('http://marnie-001-site1.atempurl.com/api/Route', {
            params: {
                from: from,
                to: to,
                startTime: datetime
            }
        }).then(function (response) {
            // navigateNext();
            console.log(response);
            var res = JSON.parse(response);
            this.setState({data: res});
            Alert.alert(JSON.stringify(response));
            // return JSON.stringify(response);
```

The second problem we had was that we couldn’t parse the response we got from our API, to a UI element, as it was not visible outside of the scope.



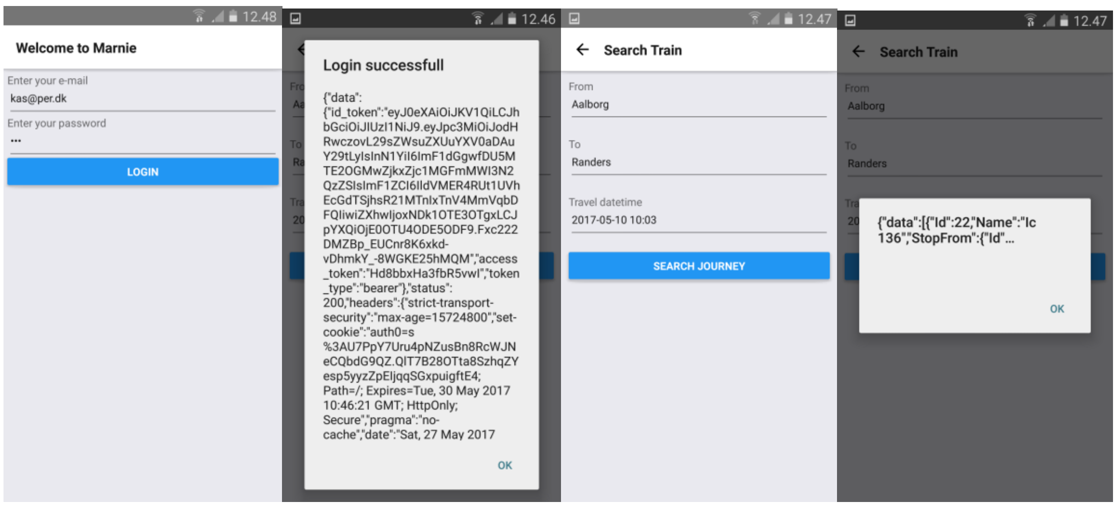
```
            var res = JSON.parse(response);
            this.setState({data: res});
            Alert.alert(JSON.stringify(response));
            // return JSON.stringify(response);
        }).catch(function (error) {
            Alert.alert('Failed', JSON.stringify(error));
            console.log(error);
        });

    return (
        <View>
            <View style={styles.button} >
```

We researched to find a possible solution, and what we found out was that the best practice to making a API call in React Native is to place them in a “componentDidMount” or a “componentWillMount” methods, because they are automatically called by React Native, and in this case, you can also access the class properties and assign the API response to them. But in our case, there is no time for the user to enter his search criteria’s, as the method will be invoked immediately after/before the search screen is loaded, and this doesn’t make sense in our app.

The third problem we encountered was the ListView component is very good at displaying a simple list of strings or numbers, but when you start to make the data structure more complex, like using a list of objects, it becomes very difficult to handle. We tried to do this, but gave up in the end, we then did some research and found that there are better third-party components for what we wanted. Specifically, the List and List item component. We tried these out by following a tutorial online [Ref 13], but unfortunately with no success. Every time we complied we got an error coming from the third-party components. Also, the guide that we tried to follow used a hardcoded array of objects as the data source, and not a JSON response, so this approach couldn't help us with the parsing.

We tried getting help from our React Native teacher, but he couldn't help us with these problems. His advice was to start from scratch with a totally new project, and design the app in another way using Redux. To be short, Redux is a technology for JavaScript apps that allows to keep the state of the app in one place. It is a container object that consists of all classes of the app and can be accessed globally. This is a huge topic and require a lot of time to dive into and understand how it works and how to use it, and we decided not to do this, because of time constraints.
There are few screenshots of what we managed to implement in React Native.



Enter your e-mail
kas@per.dk
Enter your password
•••

Login successfull

```
{"data":
{"id_token":"eyJ0eXAiOiJKV1QiLCJh
bGciOiJIUzI1NiJ9.eyJpc3MiOiJodH
RwczovL29sZWsuZXUuYXV0aDAu
Y29tLyIsInN1YiI6ImF1dGgwfDU5M
TE2OGMwZjkxZjc1MGFmMWI3N2
QzZSIsImF1ZCI6IldVMER4RUt1UVh
EcGdTSjhsR21MTnlxTnV4MmVqbD
FQIiwiZXhwIjoxNDk1OTE3OTgxLCJ
pYXQiOjE0OTU4ODE5ODF9.Fxc222
DMZBp_EUCnr8K6xkd-
vDhmkY_-8WGKE25hMQM","access
_token":"Hd8bbxHa3fbR5vwl","token
_type":"bearer"},"status":
200,"headers":{"strict-transport-
security":"max-age=15724800","set-
cookie":"auth0=s
%3AU7PpY7Uru4pNZusBn8RcWJN
eCQbdG9QZ.QIT7B28OTta8SzhqZY
esp5yyzZpEljqqSGxpuigftE4;
Path=/; Expires=Tue, 30 May 2017
10:46:21 GMT; HttpOnly;
Secure","pragma":"no-
cache","date":"Sat, 27 May 2017
```

From
Aalborg
To
Randers
Travel datetime
2017-05-10 10:03

From
Aalborg
To
Randers

```
{"data":[{"Id":22,"Name":"Ic
136","StopFrom":{"Id"...
```

In general, the time we have spent with React Native hasn't been good. There is a big lack of good tutorials and documentation to learn about React Native. The official React Native documentation consists of very basic guides and examples, that doesn't help you when you try to do more advanced things. Another thing about developing in React is that you need a good JavaScript base to really get going, and none of the group members has this.

Summary of learning process

The main goal of this project is to find out how one can learn and gain new knowledge. The best way of learning can vary from person to person. It can also depend on how one perceives information the best way - for example by watching videos, listening, reading ect or combining different approaches. As we are four in the group, we will sum up what was the best way to learn for each group member based on the experience from this project.

Christian

There are two ways that Christian learns from, and it depends on how much he knew before starting to learn something new. If he has never heard about the thing before, then he likes to watch a video and get a rundown of what this new thing is, after this he will go out and find more material like looking at guides, that could be text or video, or by reading the documentation. If Christian already knows something about the subject he prefers to go out and find text documentations and guides, as he finds this to be the fastest and most efficient way to gain the knowledge that he needs. Christian has found out throughout the years that he isn't a big fan of physical books, he prefers to read on a screen as he find that to be more comfortable, he does however like to have a lesson or course, where somebody presents something new that he needs to learn, and likes this way of learning, as it is something Christian finds challenging.

Oleksandr

It is difficult to define the best way of learning for Oleksandr, it depends on how difficult the subject is to understand for him and how comprehensive it is. In general, Oleksandr perceives information the best way by reading and following "how to" guides or tutorials and getting hands on it. If the learning area is difficult to understand or completely new for him, then he needs to supply tutorials with relevant technical documentation and code examples to understand how things work.

Watching videos is also good for him when he needs to get general idea about learning area or to understand if the approach from the video meets his needs. Following video guides is not the best way for him due to taking too long time to complete a task. He has to go back very often again and again to refresh his memory of what was going on in the video a few minutes ago.

A very good way to learn for Oleksandr is to follow a course and making a project on the side, where he implements things he has learnt in a chapter step-by-step. For example, the app for a plumber company, which we developed during Xamarin course and implemented functions as we learned them during the course.

Martin

When refreshing something old, Martin likes to look at code he wrote himself, as recognizing it helps refreshing the memory. For learning new things video with full code examples are preferred as debugging through working code gives the best understanding of how it is built. API and other documentation are used as well, to supplement examples.

Marius

When it comes to learning something new, Marius tries to use the best resources available. It might be a book, a video, courses on Udemy or any step-by-step tutorials that also provide code samples. It mostly depends on how much time is available. He might use books for bigger subjects that take a longer time to learn, like the C# programming language or, on the other side, skipping videos for subjects that don't require too much research.
Another part of Marius's learning process is based on fixing bugs and trying to put together code from different sources as there might be tutorials for creating a certain functionality but when it has to be put together in a bigger project, different errors might appear. That is why Stackoverflow or any other forums are considered to be of great help for him.

Changes to the learning goals

As we got further into the project and started working on the app, a lot of our learning goals didn't fit any more, they either got left out of the project because of time constraints, or because of some other reasons. We will highlight the changes we would make to the learning goals here, and why they changed.

Rejseplanen

Through the project, the learning goals that we set out with at the start, have changed because of problems we encountered doing the project, some were because we didn't do enough research beforehand.
The first couple of learning goals we have, mention Rejseplanen a lot, before we started the project we found the documentation for the Rejseplanen API, and thought that it would be something that we could use for a lot of things. We then figured out, a day after we started working on the project, that we would have to be approved to use their API, we applied and got a mail back instantly where it said it could take up to a week before we get an answer. We decided to wait a few days to see if we could get access. After we waited those days we had a meeting with our supervisor, where we mentioned it, and asked if it was okay to build our own Rejseplanen, he gave us thumbs up to do this. We then dropped Rejseplanen completely from the project.
We thought later in the project that we might be able to use Rejseplanen after all to find the nearest station, and we thought we might as well try to use Rejseplanen, as we got approved in the meantime. We thought before the project started that they offered this through their API, but when we tried to do it, it didn't, so we had to build this by ourselves as well.
So instead of using Rejseplanen we learned to build this service ourselves, we did this by doing research on how to compare the GPS coordinates the phone had with all train stations in Denmark.

If we were to write the learning goals today, we would change all the Rejseplanen things to mention our own API that we ended up building.

Chat

In the beginning when we brain stormed for what features the app should have, we came up with the idea of having a chat function in the app, but after doing the research on how to build a chat function and evaluating the time we had to create the feature contra more important features, we decided the best thing would be to add it to the “nice to have” feature list, as we would only create this feature if time permitted it.

Synchronization of local mobile database with the remote database

The original idea for the app was to make it possible to view some content of the app offline, by using a local database on the phone, and then synchronize the database with a remote one, when the app was in online mode. During further discussion, it became clear to the group, that this idea would not make any sense, as the group wanted to put most of the functionality on the web API, to save on battery power. We wanted to make the experience better, by always having the newest information, and if the app had an offline mode, it could confuse the users, because he would never know if the information is new or old.

User’s profile pictures

The idea to add pictures of the users appeared later during developing process and we considered it as reasonable feature of our app, since it would help the user to decide whether to send a date invitation or not. To implement this feature, we had to make some research and learn how to handle user’s pictures and where to save them.

Conclusion

The aim of this project was to develop a mobile app for train dating as mentioned in our problem statement. But also, how the group gains new knowledge through the learning progress, when solving different tasks. The group behind the project, has managed to make the app which gives the possibility to find people registered to a particular journey who wants to date. During the process, we learned a lot about how to implement different features of the app and how to solve the problems we met on the way by watching videos, following guides and tutorials, reading technical documentation and books. In the end, each group member found their own learning technique that was the best for them when trying to meet our learning goals.
The developed app has the main functionality, but it is not ready for the market as it was not the purpose of this project. To make it ready to be published for official use, some improvements have to be made to meet requirements of Google Play and App Store and

implement missing “nice to have” features. For this project, we used a free hosting with limited resources which we found suitable for study purposes, but for production version of the app, our backend should be hosted in another place.

Reference list

1. Facebook

https://channel9.msdn.com/Blogs/MVP-Windows-Dev/Login-with-Facebook-in-Xamarin-Forms
https://developers.facebook.com/docs/facebook-login/manually-build-a-login-flow

2. Building the backend and web API

http://www.tutecentral.com/restful-api-for-android-part-1/
http://www.c-sharpcorner.com/uploadfile/97fc7a/webapi-restful-operations-in-webapi-using-ado-net-objects-a/

3. Finding the nearest station

https://www.nuget.org/packages/Xam.Plugin.Geolocator
https://blog.xamarin.com/geolocation-for-ios-android-and-windows-made-easy/
http://stackoverflow.com/questions/6544286/calculate-distance-of-two-geo-points-in-km-c-sharp
https://msdn.microsoft.com/en-us/library/system.device.location.geocoordinate(v=vs.110).aspx

4. Saving login status

https://codemilltech.com/persist-whatever-you-want-with-xamarin-forms/

5. Auth0

https://auth0.com/docs/api/authentication
https://auth0.com/docs/api/management/v2

6. Web API security

https://approov.io/blog/mobile-api-security-techniques-part-1.html
https://www.codeproject.com/Articles/1005485/RESTful-Day-sharp-Security-in-Web-APIs-Basic

7. Authentication and authorization.

https://www.codeproject.com/Articles/1005485/RESTful-Day-sharp-Security-in-Web-APIs-Basic
https://approov.io/blog/mobile-api-security-techniques-part-1.html

8. SQL injection

https://www.w3schools.com/sql/sql_injection.asp

9. Native Android

https://auth0.com/docs/quickstart/native/android/getting-started

10. The Volley networking library

https://code.tutsplus.com/tutorials/an-introduction-to-volley--cms-23800

11. Using the Gson library

http://blog.takipi.com/the-ultimate-json-library-json-simple-vs-gson-vs-jackson-vs-json/

12. Using a Custom ArrayAdapter
http://www.vogella.com/tutorials/AndroidListView/article.html

13. React Native prototyping, using <List> and <ListItem>
https://hackernoon.com/getting-started-with-react-navigation-the-navigation-solution-for-react-native-ea3f4bd786a4

14. Localization
https://developer.xamarin.com/guides/cross-platform/advanced/localization/