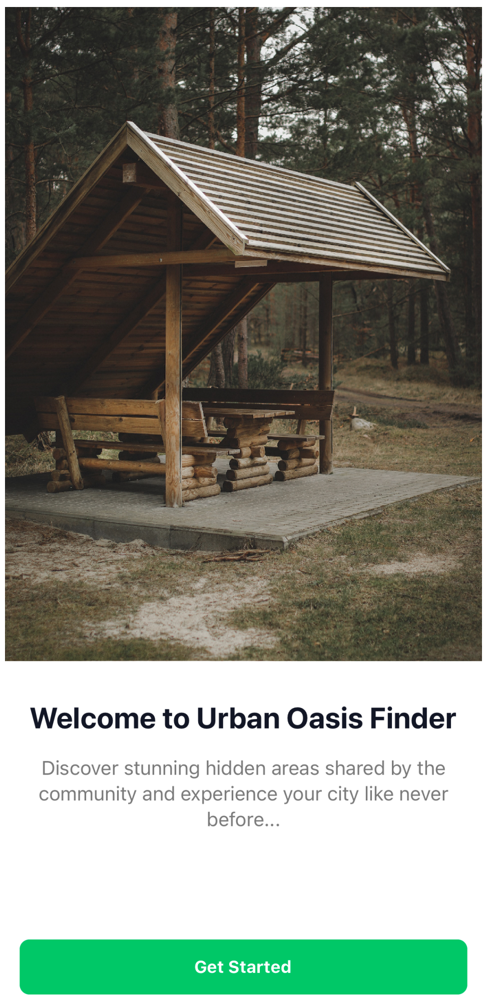
Group 10

Urban Oasis



Welcome to Urban Oasis Finder

Discover stunning hidden areas shared by the community and experience your city like never before...

Programming Mobile Applications

Course code - KAPRMOA1KU

Mikkel Q. Thynov
thyn@itu.dk

Fabian A. C. Harlang
faha@itu.dk

Iga A. Waclawska
igwa@itu.dk

IT-University of Copenhagen – Autumn 2023

Indholdsfortegnelse

1 Introduction

Exploring urban life in cities offers great experiences for tourists and locals. However, in bigger cities, it can be difficult to navigate and find areas that suit every need. In Denmark, tourism is thriving, and foreign tourists are one of the biggest markets in Scandinavia (Visit Denmark, 2022). To avoid overcrowding in the same locations and allow locals and tourists to share the city appropriately, it might be beneficial to be able to find knowledge on locations that are not considered big tourist attractions.

The green areas of cities offer a wide range of benefits (Hansen-Møller et al., 2011). Areas like parks, public gardens, forests, and other open spaces provide attractive environments for outdoor activities. By providing beautiful green spaces and meeting places for the population and encouraging them to engage in these areas, their physical and mental health can be significantly improved. Green areas also entail opportunities for social and intercultural contact, as well as spaces where urban residents and tourists can experience and learn about nature. These aspects have economic benefits, as they can attract new investments, draw tourists, and contribute to reducing public health expenditures.

Additionally, green spaces offer benefits for environmental sustainability (Brauer et al., 2016). The UN has developed 17 Sustainability Goals and this project endeavours to accommodate for UN's Sustainable Development Goals (USDG) 3 and 11, which corresponds to *Good Health and Well-being* and *Sustainable cities and communities*. Many initiatives have sought to promote sustainability and influence people's behavior to comply with USDG. Some initiatives include the use of technology and mobile applications, which are generally believed to be a viable options when trying to influence behavior. Therefore, it is relevant to examine the potential of mobile applications for encouraging tourists and locals to find and exploit.

This project seeks to develop a mobile application for finding nearby green areas in urban areas of larger cities. We label these areas as *oases*. Finding nearby and more undiscovered oases can help users to be more explorative and sustainable in the cities they visit.

2 Concept

Urban Oasis is a community-based app with the purpose of helping tourists and locals explore "hidden" locations in larger cities i.e. green oases to visit. An oasis is a green space in urban areas that includes nature in every form. This means that they can avoid classic tourist attractions which are often overrun by people and instead find local gems that might not be obvious to the general public. The users can explore areas near their current location that other private users have added via the application. The application provides a map that allows the user to explore by controlling the map view (see Figure 5) or accessing pins representing an oasis or dragging them around if they are e.g. misplaced. If they come across an oasis that they like, they can add it to the community database. If they find an oasis, they can see details of the location and press the Take me there!button to get directions from the current location.

The app provides authentication (see Figure 2, Figure 4 & Figure 3), so the user will have to create a profile from which they will have functionalities like account page, favourite oases (see Figure 6) and a list of recently visited locations (see Figure 7). When the user navigates to an oasis that another user has created, he can see details, images, and distance from his current position. If he finds the oasis interesting, he can add it to favourites by pressing the star or he can press the "Take me there!" button and find directions straight away (see Figure 8). Once the user has shown interest by accessing the details and navigating to it, the oasis will be present on the "Recently visited" page. This provides the user with the possibility to always find his previous locations that might have been particularly interesting for him.

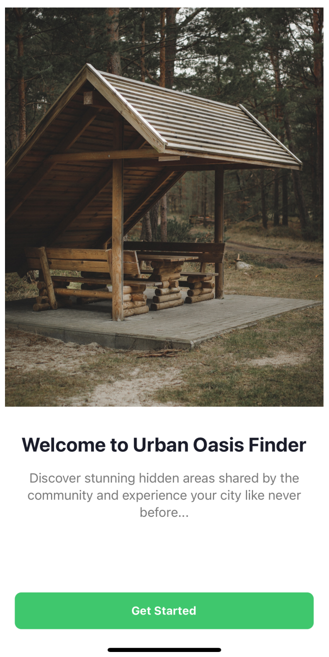
Lastly, the user can support the community by navigating to the "Add location" screen (see Figure 9 & Figure 10), where it is possible to add a new location to the database. The user can access the camera on their device and take a photo that will be added to that specific location. Furthermore, they can include a title and description and either use their current position or manually enter the address. When all information is entered correctly the new oasis is added to the community. As a result, the users can share locations based on experience and help each other to get more out of their city.

In the following figure, the QR codes for iOS and Android are presented. They are linked to the published app on Expo (see Figure 1).

(a) QR for iOS

(b) QR for Android

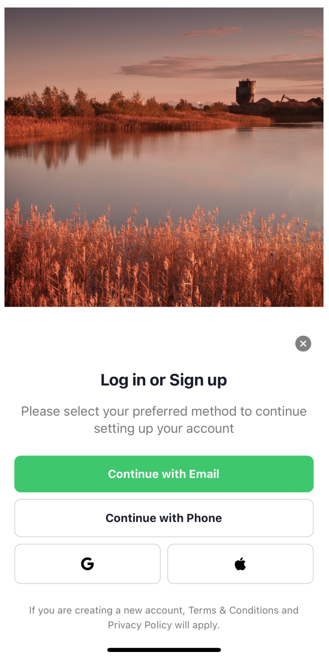
Figur 1: QR codes for our application published to the Expo server



Welcome to Urban Oasis Finder

Discover stunning hidden areas shared by the community and experience your city like never before...

Figur 2: Welcome page

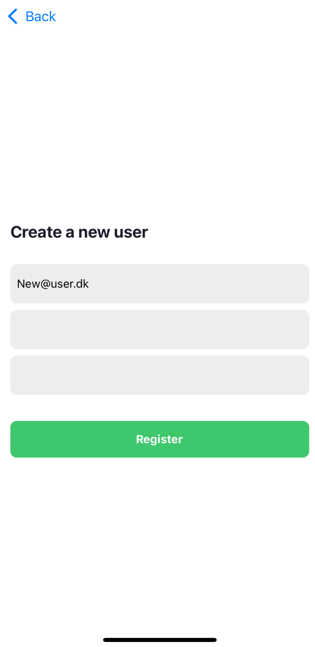


Log in or Sign up

Please select your preferred method to continue setting up your account

If you are creating a new account, Terms & Conditions and Privacy Policy will apply.

Figur 3: Login



Create a new user

New@user.dk

Figur 4: Signup

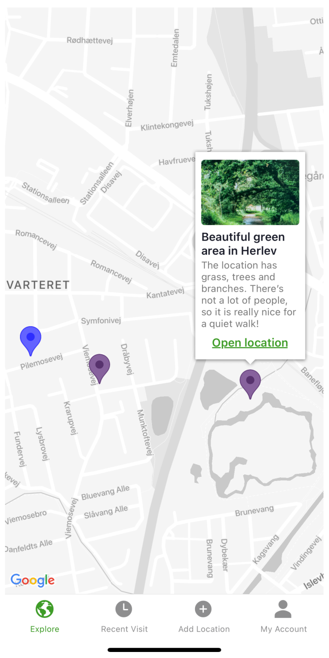
2.1 Scenario of use

Imagine a scenario where the sun is shining on a warm day during spring. You are working hard at your desk and need a break to get some fresh air. However, You are in the city where noise and pollution are filling the streets. You know the usual spots to go, same as everyone else, but you want to see a new place that is more secluded.

This is where Urban Oasis comes into play. So you pick up the phone and open the app. You sign up to create your profile and start exploring the map that displays your current location. While exploring you can see a few places nearby that look interesting from the small thumbnail image and description. You find a very interesting oasis, so you press and get to a page displaying images and details. This is a place that another user has added and recommends. It has everything you need, so you press the "Take me there!" button to get directions. The map displays the shortest distance and you navigate to the green area and enjoy a nice break at a bench in the sun. If this is a new favourite location, you add it to your list of favourites that you can always find later.

In this scenario, locals living in the city can use the app to find secluded places in their city that other people have good experiences with. Whether they need breaks at work or picnic with friends, it comes in handy to learn about less well-known places.

In a different scenario, we imagine a pair of friends backpacking in Denmark for a longer stay. They want to explore a more authentic view of Copenhagen and not visit the classic tourist attractions. To find areas that are not overcrowded they pick up the phone and open the app. Then they follow the same procedure and a nice location that matches their needs.

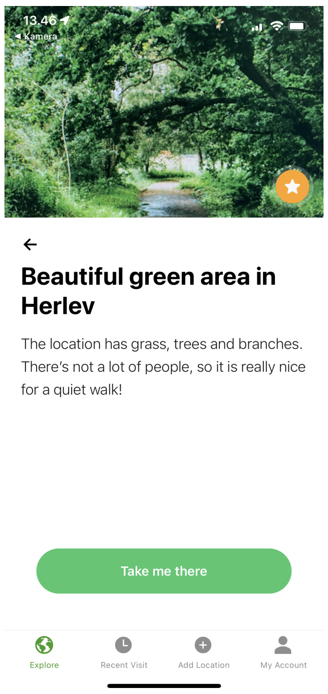


Beautiful green area in Herlev

The location has grass, trees and branches. There's not a lot of people, so it is really nice for a quiet walk!

Open location

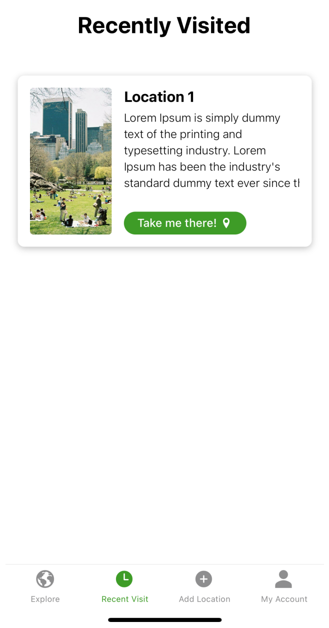
Figur 5: Explore map



Beautiful green area in Herlev

The location has grass, trees and branches. There's not a lot of people, so it is really nice for a quiet walk!

Figur 6: Location details



Recently Visited

Location 1

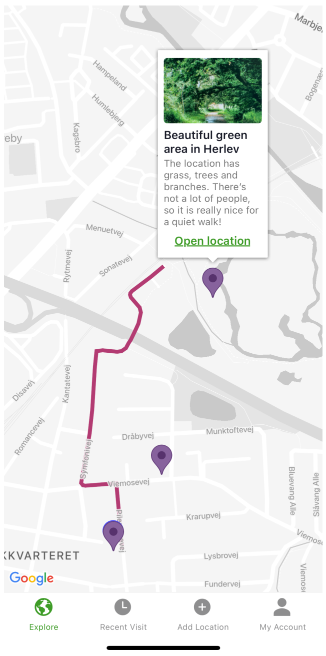
Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since tl

Figur 7: Recently visited

3 Related work

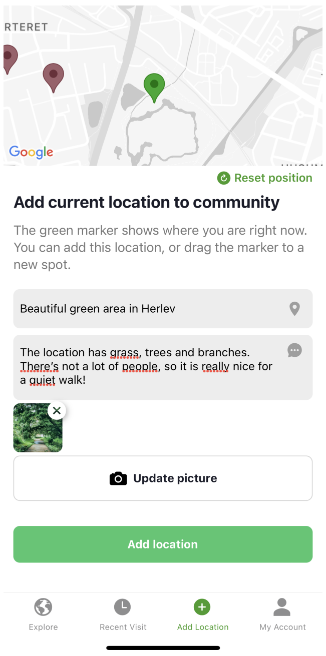
Our main inspiration was found in an app called Park4Night (P4N. It is also community-based and has many similar features ((park4night, nd)). P4N is made for motorhomes and campervans traveling on the road without knowing where to park, sleep, bath, or even what to see. Similarly, private users add locations around Europe (see Figure 12) and traveling users can see what has been suggested, uploaded, and reviewed nearby (see Figure 11). When a user finds an interesting spot, they can access it and read about the details of it (see Figure 13). They can see pictures, facilities, and reviews so they can make a qualified decision before going there.

The app is very successful and has been a big inspiration in terms of what features were needed and what design and layout the users are familiar with. Especially the functionality of the map was inspiring and adding new locations to the community. Allowing private people instead of companies and businesses, it is possible to find cheaper, less overcrowded, and more authentic places to park with a motorhome. Similarly, we want to incorporate this into Urban Oasis and allow private people to share great authentic experiences.


Beautiful green area in Herlev

The location has grass, trees and branches. There's not a lot of people, so it is really nice for a quiet walk!

Figur 8: Getting directions

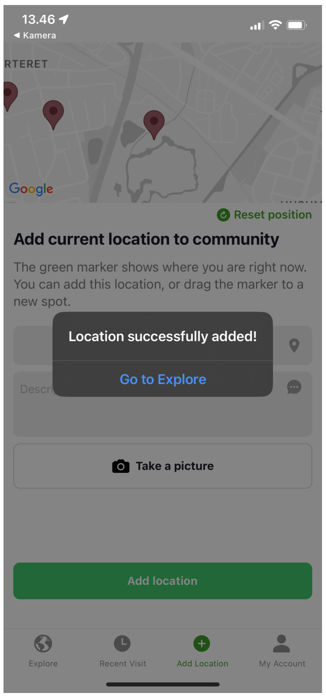

Add current location to community

The green marker shows where you are right now. You can add this location, or drag the marker to a new spot.

Beautiful green area in Herlev

The location has grass, trees and branches. There's not a lot of people, so it is really nice for a quiet walk!

Figur 9: Adding an oasis


Add current location to community

The green marker shows where you are right now. You can add this location, or drag the marker to a new spot.

Location successfully added!

Figur 10: Adding complete

4 Process

4.1 Brainstorming and sketching

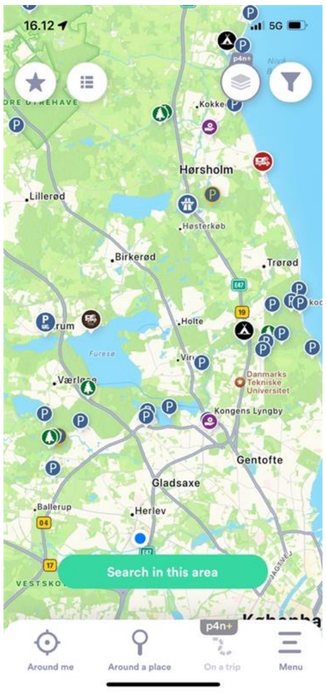
Our process began with a brainstorming based on the 4 topics from the USDG: *Good health and well-being*, *Quality education*, *Sustainable cities and communities*, and *Climate Action*. We came up with 9 ideas in total using FigJam as our platform for structuring our brainstorm (see Figure 14).

After a thorough discussion, we decided on 3 concepts to explore in more detail. All 3 concepts were visualized in Figma for a more qualified discussion. The three ideas chosen were called *Urban Oasis* (see Figure 15), *Electronics share* (see Figure 16) and *Scavenger Hunt* (see Figure 17). Based on the requirement and the concept with the most potential we agreed on the Urban Oasis concept. This idea was then implemented in more depth in Figma. The link can be found in the following.

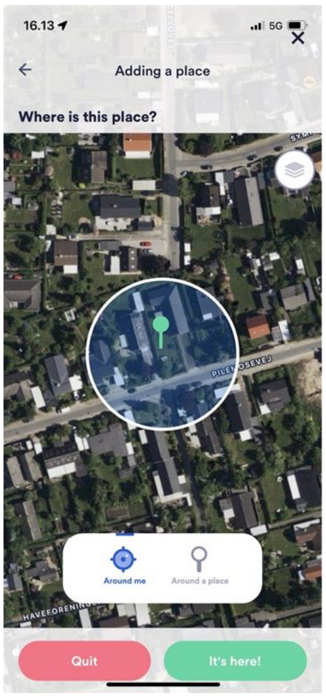
Figma link: www.figma.com/urban-oasis

4.2 Development process

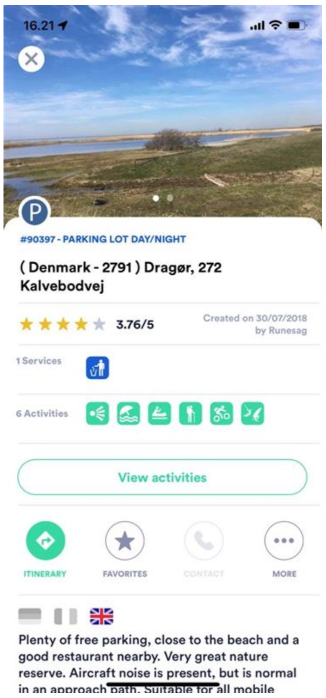
We started our implementation process by setting up a GitHub repository and decided to use the GitHub issues page as our planning platform. Here we could use a Scrum-like approach and delegate tasks to each member of the group. The development of the main parts of the app required us to carefully plan our implementation process. In consequence, we created 7



Figur 11: P4N - map



Figur 12: P4N - add


16.21

#90397 - PARKING LOT DAY/NIGHT

(Denmark - 2791) Dragør, 272 Kalvebodvej

★★★★☆ 3.76/5 Created on 30/07/2018 by Runesag

1 Services

6 Activities

View activities

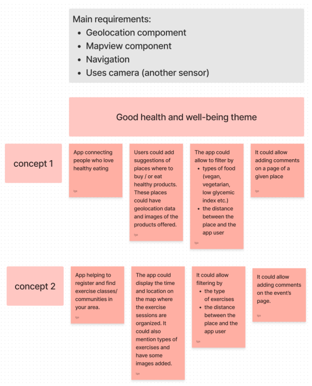
ITINERARY FAVORITES CONTACT MORE

Plenty of free parking, close to the beach and a good restaurant nearby. Very great nature reserve. Aircraft noise is present, but is normal in an approach path. Suitable for all mobile

Figur 13: P4N - details

main milestones for our entire development period (see Figure 18. These milestones included dates and specific issues that cut down each milestone into small, precise tasks to work on. Firstly, the authentication needed to be set up along with the global app navigation. This way we could display different screens depending on whether the user is logged in or not. While a group member was working on this, the rest started researching and making mock-up code to speed up implementation time when the first milestone was completed. After this, the global navigation and the bottom navigation were done, so we could begin building on the different pages. At this point, everyone could work simultaneously on tasks in the app. However, we needed good and clear communication to avoid merge conflicts from working on the same code. This was one of our strong sides since very few complicated merge conflicts appeared. Apart from that, we had some other tasks that needed to be carefully planned as a lot of functionality was built on top of the map. This includes markers, getting directions, and adding images captured from the camera. Consequently, we had to delegate other tasks as one group member was working on the map.

Important components We decided on some important components for our app. These included *Map.js*, *CustomMarker.js*, *LocationContainer.js* and the *TakeMeThereButton.js*. The *Map.js* and *CustomMarker.js* components are closely connected and we wanted to be able to add as many markers as possible to the map. Both components are displayed in our Explore screen and our Add location screen. The *LocationContainer.js* component was treated as an object holding data for each specific location that we could connect to markers on the map and a list on the Recently visited screen. The locations would also be connected to favorites



Main requirements:
- Geolocation compoment
- Mapview component
- Navigation
- Uses camera (another sensor)

Good health and well-being theme

concept 1

App connecting people who love healthy eating

Users could add suggestions of places where to buy / or eat healthy products. These places could have geolocation data and images of the products offered.

The app could allow to filter by
- types of food (vegan, vegetarian, low glycemic index etc.)
- the distance between the place and the app user

It could allow adding comments on a page of a given place

concept 2

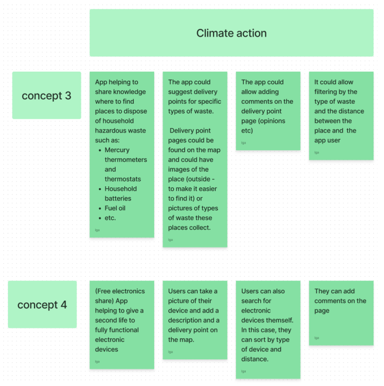
App helping to register and find exercise classes/ communities in your area.

The app could display the time and location on the map where the exercise sessions are organized. It could also mention types of exercises and have some images added.

It could allow filtering by
- the type of exercises
- the distance between the place and the app user

It could allow adding comments on the event's page.

(a) Ideas related to *Good health and well-being*



Climate action

concept 3

App helping to share knowledge where to find places to dispose of household hazardous waste such as:
- Mercury thermometers and thermostats
- Household batteries
- Fuel oil
- etc.

The app could suggest delivery points for specific types of waste.

Delivery point pages could be found on the map and could have images of the place (outside - to make it easier to find it) or pictures of types of waste these places collect.

The app could allow adding comments on the delivery point page (opinions etc)

It could allow filtering by the type of waste and the distance between the place and the app user

concept 4

(Free electronics share) App helping to give a second life to fully functional electronic devices

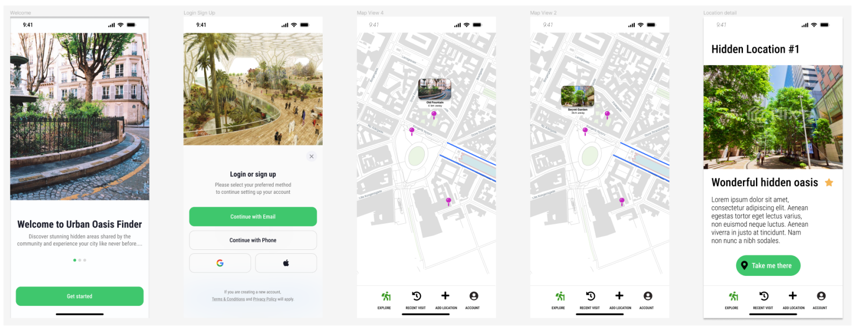
Users can take a picture of their device and add a description and a delivery point on the map.

Users can also search for electronic devices themself. In this case, they can sort by type of device and distance.

They can add comments on the page

(b) Ideas related to *Climate Action*

Figur 14: excerpt from idea generation in FigJam

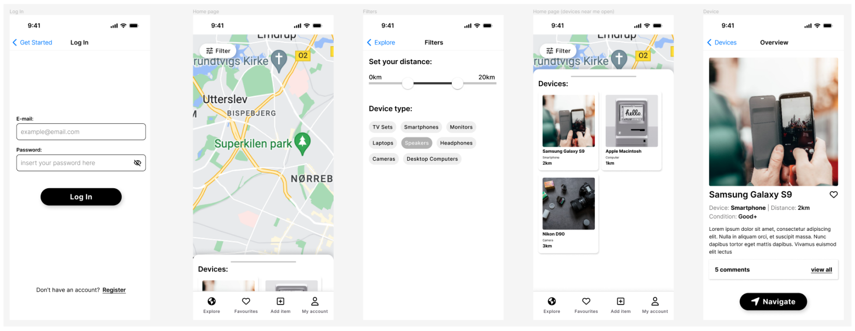

Welcome
9:41
Welcome to Urban Oasis Finder
Discover stunning hidden areas shared by the community and experience your city like never before....
Get started
Login Sign Up
Login or sign up
Please select your preferred method to continue setting up your account
Continue with Email
Continue with Phone
If you are creating a new account, Terms & Conditions and Privacy Policy will apply.
Map View 4
Old Fountain
4 km away
Map View 2
Secret Garden
3km away
Location detail
Hidden Location #1
Wonderful hidden oasis
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean egestas tortor eget lectus varius, non euismod neque luctus. Aenean viverra in justo at tincidunt. Nam non nunc a nibh sodales.
Take me there
EXPLORE
RECENT VISIT
ADD LOCATION
ACCOUNT

Figur 15: Excerpt from Urban Oasis idea in Figma

for future works. We also created a component for *TakeMeThereButton*. This was due to more consistency since we needed to use the button multiple times in the app.

4.3 Application overview

Our 2 most important screens are Explore (see Figure 15) and Add location screens (see Figure 19) because they make up the main functionality of the app. They are also the primary purpose of the app. Similarly, the authentication screens (see Figure 15) are crucial for the user, so he can control his account and store his data for the next time he visits the app. The Location details screen (see Figure 15) was also highly prioritized for the app so the user can interact with locations and read about them. Without this screen, the app would not be coherent since the map and locations would not be connected, which in consequence would result in a bad user experience. We decided to implement camera support and directions (see Figure 19) to a location as they are also part of the core functionality. These were quite complicated and time-consuming but were prioritized as "need to have" for our project. On the other hand, 2 screens had lower priorities. These included the Recently visited page and the favorites page, which we planned to implement on the account page. The Recently visited page was created later in the process whereas the favorites page was excluded from the app.



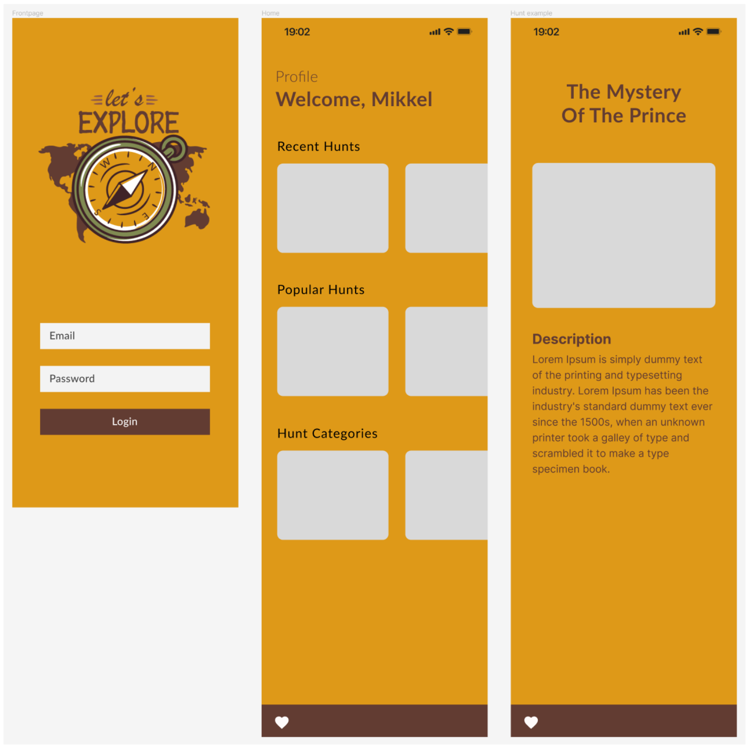
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla in aliquam orci, et suscipit massa. Nunc dapibus tortor eget mattis dapibus. Vivamus euismod elit lectus

Figur 16: Excerpt from Electronics Share idea in Figma

This was due to time pressure and the fact that favourites were prioritized as ”nice to have”.

Geolocation Geo-location implementation was an obvious must-have for our applications since the core of the app is based on exploring locations in the user's proximity. Conveniently, a feature-rich map library/component *react-native-map* is built for React Native and can, with a few properties, initialize a map. Following the lectures and the documentation, we discovered how to request permission to access geolocation from the user's device and start fetching coordinates from the user. Using latitude, longitude, and delta values an initial region can be set or changed more dynamically based on user interaction. We also added some hard-coded locations and displayed them using <Markers> to start playing around with the map functionalities. Subsequently, using them more dynamically, we added user-contributed locations that could be added with images, descriptions, and coordinates that would correspond with the location of the user submitting them. This allows other users in the community to request directions and be guided there with ease. (using Google API).
For easier development, we decided to create a context for our applications (*locationContext.js*) allowing state variables to be shared globally in our project so user coordinates, location coordinates, etc. can be accessible within different components. Additionally, we managed to style the map to match our Figma prototype using an online styling utility tool *(https://mapstyle.withgoogle.com/)*. The specific components relevant to geolocations will be specified and elaborated further on in our contribution sections.

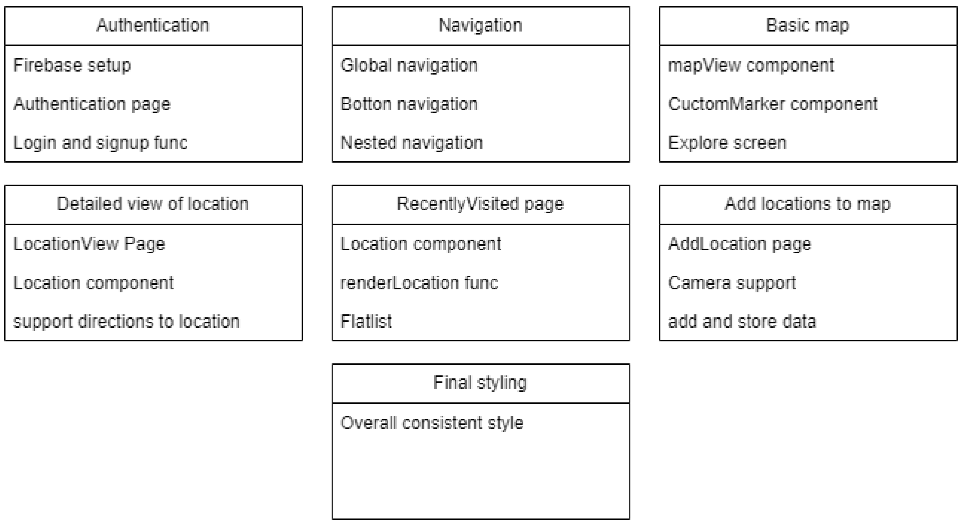
Authentication Authentication implementation was necessary for security reasons. It prevents unauthorized people from accessing the personal accounts of potential Urban Oasis users. Authentication services implemented in the application are provided by Firebase. Users who wish to be authenticated can decide what action to take when on the *Log in or Sign up page*. Four options are available to proceed from this step: Continue with Email, Phone, Google, or Apple. Currently, the application supports the first option. After clicking the *Continue with Email* button, users are navigated to the *Log in* page. Once there, they are expected to type their email address and password in the input fields. It is worth noting that


Frontpage
Home
Hunt example
let's
EXPLORE
Email
Password
Login
19:02
Profile
Welcome, Mikkel
Recent Hunts
Popular Hunts
Hunt Categories
The Mystery
Of The Prince
Description
Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book.

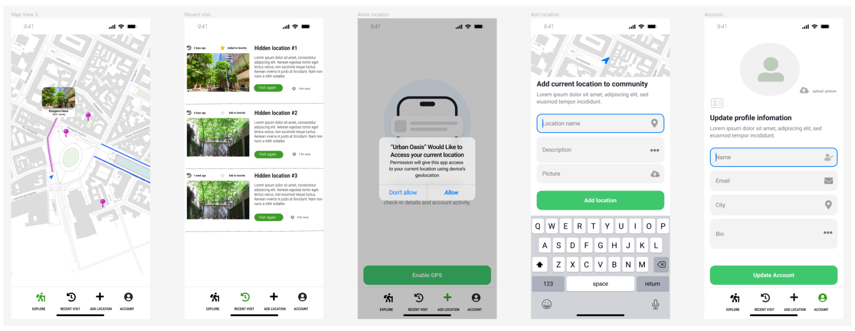
Figur 17: Excerpt from Scavenger Hunt idea in Figma

the components of the *Log in* page are placed inside the <*KeyboardAvoidingView/*> instead of a <*View/*> to prevent input fields from being covered behind the virtual keyboard. Users are granted access to the application functionalities after inserting valid login credentials and clicking the *Log in* button. If the user does not have an account, they can click the *Don't have an account? Sign up* button, which will redirect them to the Registration page. The Individual contribution section provides a more detailed description of how authentication is implemented.

Camera The core purpose of the application is to share and recommend green locations in urban areas. This made us realize that relying solely on the name and description of the place is insufficient for this task. First, it is time-consuming to write extensive descriptions, and only some users might be willing to do it. Second, descriptions can be subjective. The best solution to this problem is to allow users to take and attach pictures to the places they share. To enable this, we decided to implement Expo Camera in Urban Oasis. The functionality is simplistic, utilizes the rear and front camera of the smartphone, and can be accessed from the *Add Location* page by clicking the *Take a picture* button. More technical details on the Expo Camera implementation can be found in the Individual contribution section.



Figur 18: Overview of milestones



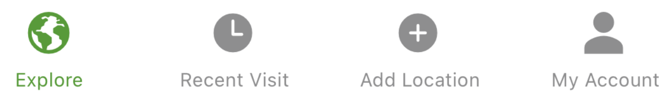
Figur 19: Excerpt from Urban Oasis idea in Figma

4.4 Feedback sessions

Discussing the current implementation at the feedback sessions gave us some valuable insights that we did not account for. First of all, we had a hard time figuring out where to place some of the functionality in the app. We were not sure where to place the favourites feature in the app and the feedback session helped us decide that the feature should be positioned on the account page. As a result, placing it here would be more intuitive for the user. Furthermore, our navigation became quite complex, so we wanted some feedback on how we could nest the navigation appropriately instead of using conditional rendering. This was discussed and a decision was made during class so we could move forward.

4.5 Design guidelines

An important aspect of our app design was using Normans HCI guidelines (Lupanda and Rensburg, 2021). Firstly, we wanted to make the user able to constantly see what actions are available in the app, hence providing proper discoverability for the user. In consequence, we decided to use bottom navigation (see Figure 20) without main features present for the user at all times. The navigation bars include signifiers i.e. clear and visible icons with a brief label. This makes it clear to the user what main actions are possible. Furthermore, when a location is added we present the user with an alert to give appropriate feedback on his action. The alert tells the user that the location is successfully added and that he can navigate to the Explore screen to view it (see Figure 10).



Figur 20: Bottom navigation

We wanted to be consistent with colors, icons, and buttons. For that reason, the "TakeMeThere!" button was implemented as a component, so we could reuse the design multiple places in the app and thus guide the user to always know that the design of this button, helps them to navigate to the location they seek. Following the principles of Johnson (Lupanda and Rensburg, 2021) we decided early on to focus on function rather than presentation. Consequently, we chose to move the milestone for our final styling to the end of our development process. Looking back at our project, we could have benefited from using more feedback in our authentication page, by providing the user with feedback if he did something wrong e.g. writing a wrong password or email when logging in. This would also take advantage of Stone's principle of Tolerance since we would prevent the user from making errors.

4.6 Individual contribution

The group consist of three group members: **Fabian**, **Iga** and **Mikkel**. The work of each group member is described in detail in this section and separated into subsections describing the work of each member.

4.6.1 Fabian

MapView The MapView-component is built using the react-native-map library and allows the user to explore locations, get directions, and display the user's current location. It, therefore, utilizes another component *locationContext* from which a function is created that requests permissions to grant access to the user's device coordinates and fetches this information and stores it in state and context for other components to use:

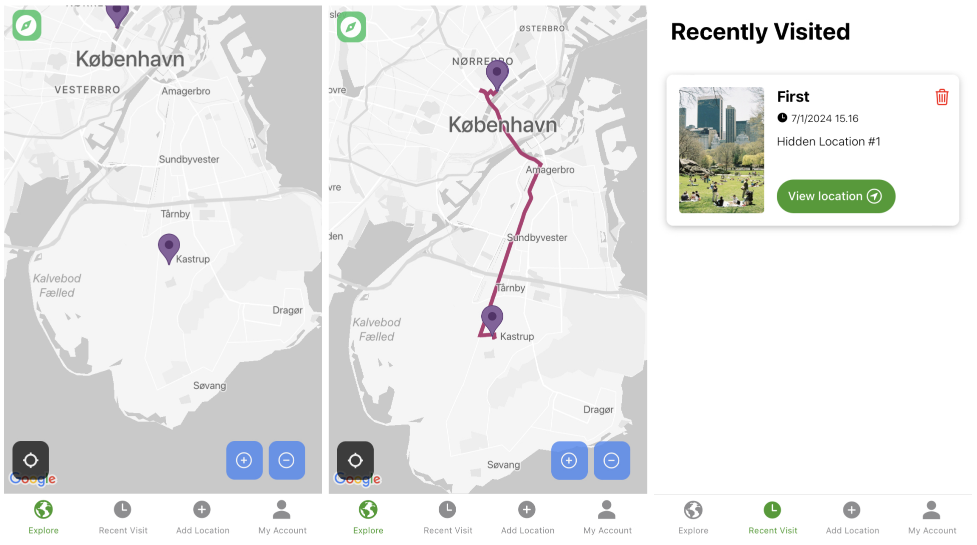
```
const requestLocationPermissionAndUpdatePosition = async () => {
    try {
      let { status } = await Location.
    requestForegroundPermissionsAsync();
      if (status !== "granted") {
        console.error("Permission to access location was denied");
        return;
    }

    // Get the current position
    let location = await Location.getCurrentPositionAsync({});
    setCurrentPosition({
      latitude: location.coords.latitude,
      longitude: location.coords.longitude,
    });
```

Once the user's location is fetched and stored the <*Mapview*> component can be rendered and properties set accordingly. The initial region is set to its current position and a low delta value creating a zoomed-in view per default suiting for an app like this where users explore locations within walking distance of cities. Markers are loaded from a data array of the added locations persisted in memory using *asyncStorage*. This array is then iterated over using *.map* function and for each item a <*CustomMarker*> is created and placed on the the map at a particular coordinate. Pressing the marker will display an image, and description inside a callout (pop-up) with an option to press the location for more details.

Moreover, we've used Google as a map provider which allows custom styling of the map's appearance using a JSON file stored in a constant and sent to the *CustomMapStyle* property. With the above implementation in place, the component can be rendered as follows:

```
  <View>
      <MapView
        provider={PROVIDER_GOOGLE}
        ref={mapRef}
        style={styles.map}
        initialRegion={{
          latitude: currentPosition
            ? currentPosition.latitude
            : 55.60866491013769,
          latitudeDelta: 0.004,
          longitude: currentPosition
            ? currentPosition.longitude
            : 12.5911277895021,
          longitudeDelta: 0.003,
        }}
        region={region}
        customMapStyle={mapTheme}
        gestureEnabled={true}
      >
        {/* Maps through array DefaultLocations and displays markers
 */}
        {showDefaultLocations()}
        ... </MapView>
```

For enhanced user interaction, the map view has been extended with control buttons to easily alter the displayed map. At the bottom left of the map, a *recenter-button* can be pushed to calibrate the coordinate of the map back to the current position of the user. This is useful when the user wants to go back to the current location with one push rather than multiple finger gestures on the display. Additionally, zoom-in and -out-buttons are accessible in the bottom right of the screen, offering four levels of zoom to intuitively adjust the view—whether widening the scope to discover new locations or narrowing in for a closer look at the current position as the user moves around town. Some conditions styling is added as well, grey-scaling and disabling the buttons when a certain maximum or minimum zoom level is reached. Finally, a compass is shown in the corner allowing the user to toggle between GPS functionality i.e. if the route to a certain destination should be shown on the map or not. *(figure 20a)*



(a) Control buttons (b) Directions (c) Recent Visits

Figur 21: MapView - Functionalities

Directions component - Take me There When the user 'Opens' a location a more detailed view of the instance will show, with pictures, descriptions, etc. The green 'Take me there'-button will trigger the directions function. Using context the component *(locationDetails)* has access to the user's current position and the coordinate of the desired destination. Using the Google Directions API Google Maps Platform (2024) a destination will be calculated and then stored in state and subsequently decoded to value readable by visual polyline (colored route library) and displayed on the map:

```
         <Polyline
              coordinates={polyline.decode(directions).map((point) => ({
                latitude: point[0],
                longitude: point[1],
              }))}
              strokeWidth={5}
              strokeColor="#b33b72"
```

```
            />
```

Once the destination is calculated the handleTakeMeThere-function will navigate the user back to mapView initializing the route. *(figure 20b)* The compass icon in the corner triggers an onPress-event allowing the users to toggle the route on and off. i.e. showing/hiding the coloured polylines. This is controlled using some state logic and a ternary operator. Destination holds the value given to the polyfill property. The *prevDestination* holds a copy of directions and can reinitialize the route while the original destination otherwise is set to null to clear the route. The styling of the button follows this logic and will be displayed with either low opacity or high opacity based on this condition.

```

onPress={() => {
            setPrevDirections(directions);
            setDirections(null);
            setDirections(directions === null ? prevDirections :
 null);
          }}
          style={[
            styles.clearRoute,
            { opacity: directions === null ? 0.3 : 0.9 },
          ]}
```

In addition, pushing the 'Take me there'-button invokes another event, namely that the user most likely has attempted to visit this particular location. Therefore the function also creates a timestamp and constructs an object representing the visited location, including its ID, title, description, image, location, and visit date and time. Then it updates the recentlyVisited array in state with the new location information. This array is subsequently iterated over and displayed in a list in another component 'Recently visited'. From here users can see when they visited a particular location, revisit it, or delete it, if they wish. These operations are handled by a *.filter* method that returns the updated array, which includes all locations except the deleted one, and updates its state via *setRecentlyVisited. (figure 20c)* Although the implementation of visited locations may be naive and more preferably could have utilized technology like geofencing, we decided for this project to go with it. A concern regarding geofencing is that users "accidentally" would be near a location and the app would mistakenly assume they had visited it. Ideally, a combination of geofencing and a check-in button would cut it and could be considered for further implementation. Moreover, the directions feature could be enhanced with a distance away label and more options for time estimation for walking there proportional to the current location. These improvements are suggestions, however, beyond the scope of this project and can potentially be allocated to further work and ideation. (5.3)

4.6.2 Iga

React Navigation One of my responsibilities was implementing React Navigation in the Urban Oasis app, including a Stack Navigator and a Bottom Tab Navigator.

To build the structure and hierarchy of all the navigation types, I decided to follow Fabricio's Narcizo (2023a) GitHub example from Lecture 5 as an inspiration, mainly because I found this solution very cohesive. It involved separating the navigation functionality into a separate module and turning specific navigation parts into distinct components.

On the *App.js* level, the navigation is imported as a single component and is surrounded by multiple context providers (see Listing 1).

```
export default function App() {
  return (
    <AuthenticationContextProvider>
      <CameraContextProvider>
        <LocationContextProvider>
          <Navigation />
        </LocationContextProvider>
      </CameraContextProvider>
    </AuthenticationContextProvider>
  );
}
```

Listing 1: App.js

I have included two distinct navigation components inside the aforementioned Navigation component: *<ApplicationNavigation />* and *<AuthenticationNavigation />* (see Listing 2). The authentication status of the user determines which type of navigation and screens are available to them. As can be seen in the example, this separation is possible thanks to incorporating the ternary operator and the *isAuthenticated* boolean. Given that I implemented React Navigation before Firebase Authentication, initially, I had to improvise and hardcode the *isAuthenticated* value. Later in the process, the said value was taken from the *AuthenticationContext*.

```
export const Navigation = () => {
  const { isAuthenticated } = useContext(AuthenticationContext);

  return (
    <NavigationContainer>
      {isAuthenticated ? (
        <ApplicationNavigation />
      ) : (
        <AuthenticationNavigation />
      )}
    </NavigationContainer>
  );
};
```

Listing 2: index.js from the *navigation* module

If users are not authenticated, they will see the AuthenticationNavigation, which includes the *Welcome*, *Log in or sign up*, *Log in*, and *Sign up* screens (see Listing 3). I have decided to use Stack Navigator with customized headers for these particular screens.

```
//Stack navigation screens
export const AuthenticationNavigation = () => {
  return (
    <Stack.Navigator>
      <Stack.Screen
        name="Welcome"
        options={createScreenOptionsForIntro}
        component={WelcomePage}
      />
      <Stack.Screen
        name="Log in or Sign Up"
        options={createScreenOptionsForIntro}
        component={LoginOrSignupPage}
      />
      <Stack.Screen
        name="Log in"
        options={createScreenOptionsForAuthentication}
        component={LoginPage}
      />
      <Stack.Screen
        name="Sign up"
        options={createScreenOptionsForAuthentication}
        component={RegistrationPage}
      />
    </Stack.Navigator>
  );
};
```

Listing 3: AuthenticationNavigation.js

Authenticated users will access the *ApplicationNavigation* (see Listing 4). Following the Figma prototype layout, I have implemented it as the Bottom Tabs Navigator. It consists of four tabs: *Explore*, *Recent Visit*, *Add Location*, and *My Account*. Similarly, as in Fabricio's example, I have dynamically assigned an individual Icon imported from the Expo icon library to each of the tabs. To navigate between tabs, users need to click on the tab with the name of the screen they wish to go to.

```
//Bottom Navigation tabs and screens
export const ApplicationNavigation = () => {
  return (
    <Tab.Navigator screenOptions={createScreenOptions}>
      {/* Tab screen "Explore" contains nested navigation */}
      <Tab.Screen name="Explore" component={ExploreNavigation} />
      <Tab.Screen name="Recent Visit" component={RecentlyVisited} />
      {/* Tab screen "Add Location" contains nested navigation */}
      <Tab.Screen name="Add Location" component={AddLocationNavigation
    } />
      <Tab.Screen name="My Account" component={UserAccount} />
    </Tab.Navigator>
  );
};
```

Listing 4: ApplicationNavigation.js from the *navigation* module

It is also worth noting that the *Explore* and *Add Location* screens have their own nested Stack Navigators, named respectively *ExploreNavigation.js* (implemented by Mikkel) and *AddLocationNavigation.js* (implemented by me). A description of the latter can be found in the upcoming paragraph.

Nested Navigator - *AddLocationNavigation* Initially, I was not aware of the need for such a solution. However, I realized it was necessary when working on the Camera functionality on the *AddLocation.js* page. After the *Take a picture* button is clicked, the application needs to enable access to the *Camera* page within the *Add Location* Tab. In theory, it sounded like a nested Stack navigation inside a specific Tab, but I needed to figure out how this could be achieved. To solve this problem, I reviewed the React Navigation documentation and discovered a solution that met my needs (see Listing 5).

Following the documentation guide, I came up with this solution:

- Identified all screens that should form a nested Stack Navigation within the *Add Location Tab*. (the *Add Location Screen* and the *Camera Screen*)
- Created a Stack Navigation component called *AddLocationNavigation* that contained these screens
- Imported *AddLocationNavigation* into *ApplicationNavigation.js* and inserted it as a component prop inside of the main Tab Navigator

```
the export const AddLocationNavigation = () => {
  return (
    <Stack.Navigator>
      <Stack.Screen
        name="Add Location Screen"
        options={createScreenOptions}
        component={AddLocation}
      />
      <Stack.Screen
        name="Camera Screen"
        options={createScreenOptions}
        component={CameraView}
      />
    </Stack.Navigator>
  );
};
```

Listing 5: addLocationNavigation.js from *navigation* module

Authentication Authentication is essential to ensure safety, as it prevents unauthorized access to the application. After developing the navigation, my next task was integrating Firebase authentication into Urban Oasis. To achieve it, I followed the instructions from lecture 5 and used GitHub code examples by Fabricio Narcizo (2023b) as a starting point. The listing below (see Listing 6) depicts a config code that Firebase generated after following all the steps of setting the project up using the Firebase Console.

```
// Import the functions you need from the SDKs you need
import { initializeApp } from "firebase/app";
import { initializeAuth, getReactNativePersistence } from "firebase/
    auth";
import ReactNativeAsyncStorage from "react-native";
// TODO: Add SDKs for Firebase products that you want to use
// https://firebase.google.com/docs/web/setup#available-libraries

// Your web app's Firebase configuration
// For Firebase JS SDK v7.20.0 and later, measurementId is optional
const firebaseConfig = {
  apiKey: "AIzaSyCA3rCWTw2VeercNkucylsh1YJrR72D38A",
  authDomain: "urban-oasis-a66d0.firebaseapp.com",
  projectId: "urban-oasis-a66d0",
  storageBucket: "urban-oasis-a66d0.appspot.com",
  messagingSenderId: "682647272581",
  appId: "1:682647272581:web:df98c34545483f2583fdec",
  measurementId: "G-36W1HMN2JR",
};

// Initialize Firebase
const app = initializeApp(firebaseConfig);

// Initialize Firebase Authentication and get a reference to the
    service.
export const auth = initializeAuth(app, {
  persistence: getReactNativePersistence(ReactNativeAsyncStorage),
});
```

Listing 6: firebaseConfig.js

AuthenticationService.js (see Listing 7) contains methods provided by Firebase Authentication (2024). The *createUserWithEmailAndPassword* creates a new user account with their email address and password. The s*ignInWithEmailAndPassword* lets users who already have an account created to log in with their credentials.

```
import {
  signInWithEmailAndPassword,
  createUserWithEmailAndPassword,
} from "firebase/auth";

import { auth } from "../firebaseConfig";

export const loginRequest = (email, password) => {
  return signInWithEmailAndPassword(auth, email, password);
};

export const registerUser = (email, password) => {
  return createUserWithEmailAndPassword(auth, email, password);
};
```

Listing 7: AuthenticationService.js

The *AuthenticationContext.js* (see Listing 8) file globally sets and shares authentication-related states and booleans, defines and shares functions that handle login, registration, and log-out processes for users (*onLogin, onRegister, onLogout*)

```
import { loginRequest, registerUser } from "./AuthenticationService";

export const AuthenticationContext = createContext();

export const AuthenticationContextProvider = ({ children }) => {
  const [isLoading, setIsLoading] = useState(false);
  const [user, setUser] = useState(null);
  const [error, setError] = useState(null);

  useEffect(() => {
    onAuthStateChanged(auth, (user) => {
      if (user) {
        setUser(user);
      }
      setIsLoading(false);
    });
  }, []);

    {...}

  return (
    <AuthenticationContext.Provider
      value={{
        isAuthenticated: !!user,
        user,
        error,
        isLoading,
        setError,
        onLogin,
        onRegister,
        onLogout,
      }}
    >
      {children}
    </AuthenticationContext.Provider>
  );
};
```

Listing 8: AuthenticationContext.js

Add a new location - the page Add a new location page allows adding a new, green location to the *defaultLocations* array (and to the community). The page can be accessed for authenticated users by clicking the *Add Location* tab.

As the main container on this page, I have decided to use a *<KeyboardAvoidingView />* (see Listing 9) as the main container instead of a generic <View /> to prevent input fields from being hidden behind the expanded keyboard. *<KeybordAvoidingView />* moves all the content upward. Therefore, input fields that are present on the page and can be accessed with the keyboard open.

```
export const AddLocation = ({ navigation }) => {

{...}

  return (
    <KeyboardAvoidingView
        {...}
    </KeyboardAvoidingView>
  );
};
```

Listing 9: Keyboard Avoiding View

The most essential data for creating a new location and adding a new pin to the map is its *longitude* and *latitude*. We all agreed that manually providing this information as numerical values in the input fields would be inconvenient and inefficient. The best alternative was using the *Map.js* component and its states shared in the *locationContext.js* to extract this data.

Because of the limited space on the *addLocation.js* page, I have decided to adapt the *<Map/>* component to display only the necessary items for adding a new place (see Listing 10):

- It does not display “take me there” routes.
- Existing locations are rendered but only within a <Marker/> without the <Callout/> support. Initially, I considered hiding added places entirely, but I realized that, in theory, it might lead to creating duplicated locations.

- Instead of the blue <*Marker/*> displaying the current user position, the <Map/> renders a green draggable <*Marker/*>, which by default displays the current position of the user as well but can be moved to another position. Its *longitude* and *latitude* states are used in the process of creating the location object.

```
<View style={styles.imageWrapper}>
<Map screenType={"AddLocation"} />
</View>
```

Listing 10: Map component

After incorporating a way to extract the latitude and longitude of a location, the next important step was to implement a way of naming the place and, optionally, adding some description to it. I created two local states, *locationName* and *description*, with empty strings as initial values. Each state and setter are passed as props to their respective <C*ustomInputField/*> components (see Listing 11). In there, their values are updated when the *onChangeText* event is triggered.

```
export const AddLocation = ({ navigation }) => {
  const [locationName, setLocationName] = useState("");
  const [description, setDescription] = useState("");
{...}
  return (
    {...}
          <View style={styles.inputSection}>
            <CustomInputField
              hiddenInput={false}
              placeholder={"Location name"}
              input={locationName}
              setInput={setLocationName}
              icon={
                <Ionicons name="md-location-sharp" size={20} color="#9
    e9e9e" />
              }
            />
            <CustomInputField
              hiddenInput={false}
              placeholder={"Description"}
              input={description}
              setInput={setDescription}
              isMultiline={true}
              icon={
                <Ionicons
                  name="chatbubble-ellipses-sharp"
                  size={20}
                  color="#9e9e9e"
                />
              }
            />
    {...}
  );
};
```

Listing 11: Custom input fields

It is possible to add a photo to a new location. This functionality can be accessed by clicking the *Take a picture* button (see Listing 12). When its *onPress* event listener is triggered, it executes the *clickNavigateToCamera* function (see Listing 13) which replaces the *AddLocation* page with the *Camera* page within the *Add Location* tab.

When the photography is successfully captured and saved, the state of its *URI* changes from null to a string value that allows locating and fetching it from the media of the smartphone library. I opted to use this state change to update the button text conditionally to display *Update picture* once a photo is taken (see Listing 12).

```
<CustomButton
  onPress={clickNavigateToCamera}
  value={uri ? "Update picture" : "Take a picture"}
  theme={"secondary"}
  icon={<Ionicons name="camera" size={24} color="black" />}
/>
```

Listing 12: Take a picture button

```
  const clickNavigateToCamera = () => {
    navigation.navigate("Camera Screen");
  };
```

Listing 13: clickNavigateToCamera

When testing the *Take a picture* functionality, it became clear to me that there was no indication of whether a photo was added. Moreover, there was also no option of removing it. To address these issues, I implemented a small thumbnail image preview on the *Add Location* page (see Listing 14). It is rendered conditionally and, similarly to the button's text, based on the *URI* state of the image. If the state of *URI* is falsy (equal to null), nothing is rendered on the page. In contrast, when the state of *URI* is truthy (contains a string value), a <*View*/> component with <*Ionicons*/> and <*Image*/> children is rendered. There is a small icon located in the right corner of the image thumbnail. Clicking it triggers the *clearImage* function, which runs *setUri(null)* and thus removes the attached photo.

```
export const AddLocation = ({ navigation }) => {

{...}
  const { uri, setUri } = useContext(CameraContext);

  return (
        {...}
            {uri && (
              <View>
```

```
                <Ionicons
                  style={styles.deleteImage}
                  name="md-close-circle-sharp"
                  size={28}
                  color="white"
                  onPress={clearImage}
                />
                <Image
                  style={styles.imageSection}
                  source={{ uri: uri, isStatic: true }}
                />
              </View>
        {...}
  );
};
```

Listing 14: Image preview and conditional rendering

The last component I implemented on this page is the *Add location* button (see Listing 15). Clicking it triggers the *addLocation* function.

```
<CustomButton onPress={addLocation} value={"Add location"} />
```

Listing 15: *addLocation* function called from the *CustomButton* component in the *AddLocation.js* file

The function *addLocation* (see Listing 16) is responsible for carrying out multiple tasks:

- First, to ensure the correct display of new data, the *trim* function clears unnecessary white space characters from the *locationName* and *description* states.
- After trimming, the mandatory string, *trimmedLocationName* is checked against the length condition. If its length is **not zero**, the function proceeds.
- Then, new *LocationItem* object is initialized and populated with data. While the *trimmedLocationName* is mandatory, the *trimmedDescription*, *latitude*, *longitude*, and image *URI* are optional.
- Next, a new array, *updatedLocation* is created. It contains the *defaultLocations* data and the newly created location.
- The *defaultLocations* state is updated by executing *setDefaultLocations(updatedLocations)*.
- The *updatedLocations* array is stringified and appended to *AsyncStorage*, thus turned into permanent data.
- Lastly, after the new location (oasis) is added, the locationName, description, uri and green marker position are cleared so that the page is ready for adding another location.

```
const addLocation = async () => {
    let trimmedLocationName = locationName.trim(); //cleans the input up
    let trimmedDescription = description.trim(); //cleans the input up
    let latitude = draggableMarkerCoordCurrent.latitude;
    let longitude = draggableMarkerCoordCurrent.longitude;

    if (trimmedLocationName.length !== 0) {
      let newLocation = new LocationItem(
        trimmedLocationName,
        trimmedDescription,
        latitude,
        longitude,
        uri
      );

      //appends new location object to the defautLocations array
      const updatedLocations = [...defaultLocations, newLocation];

      setDefaultLocations(updatedLocations);

      try {
        await AsyncStorage.setItem(
          "ALL_LOCATIONS",
          JSON.stringify(updatedLocations)
        );
        console.log("New location added to Async storage");
      } catch (error) {
        console.log(error);
      }
      setLocationName(""), setDescription(""), setUri(null);
      createLocationAddedAlert(), handleGreenMarkerReset();
    } else {
      Alert.alert("To add a new location, you need to provide its name");
    }
  };
};
```

Listing 16: Add location function

Add a new location - camera support To implement camera functionality in the Urban Oasis app, as a starting point, I have used the example of the camera service and camera context shared on GitHub repository by Fabricio Narcizo (2023c), under lecture 10.

Like in the camera implementation shared by Fabricio Narcizo, The CameraService in Urban Oasis (see Listing 17) contains definitions of the following functions:

- The *hasCameraPermission* function that takes care of all the system permissions to access the camera and media library.

- The *snapAndSavePhoto* function that takes care of taking a picture and saving it to the media library of a device.

```
import { Camera } from "expo-camera";
import * as MediaLibrary from "expo-media-library";

export const hasCameraPermission = async () => {
  let { status: cameraStatus } = await Camera.
    requestCameraPermissionsAsync();
  if (cameraStatus !== "granted") {
    console.log("Permission to access the camera was denied.");
    return false;
  }

  let { status: mediaStatus } = await MediaLibrary.
    requestPermissionsAsync();
  if (mediaStatus !== "granted") {
    console.log("Permission to access the media library was denied.");
    return false;
  }

  return true;
};

export const snapAndSavePhoto = async (camera) => {
  const albumName = "PMA 2023";
  const { uri } = await camera.takePictureAsync();
  const asset = await MediaLibrary.createAssetAsync(uri);
  let album = await MediaLibrary.getAlbumAsync(albumName);

  if (!album) {
    await MediaLibrary.createAlbumAsync(albumName, asset);
  } else {
    await MediaLibrary.addAssetsToAlbumAsync(asset, album);
  }

  return await MediaLibrary.getAssetInfoAsync(asset.id);
};
```

Listing 17: cameraServices.js

Similarly, the *CameraContext* (see Listing 18) is responsible for:

- Executing functions from the *CameraService.js*.
- Globally sharing functions for camera - toggleCamera and snapAndSave.
- Globally sharing camera states such as hasPermission, camera, type, URI, and their setState() methods.

```
import React, { createContext, useEffect, useState } from "react";
import { CameraType } from "expo-camera";

import { hasCameraPermission, snapAndSavePhoto } from "./cameraServices.js";

export const CameraContext = createContext();

export const CameraContextProvider = ({ children }) => {
  const [hasPermission, setHasPermission] = useState(false);
  const [camera, setCamera] = useState(null);
  const [type, setType] = useState(CameraType.back);
  const [uri, setUri] = useState(null); //uri of all images taken and saved to gallery

  useEffect(() => {
    (async () => {
      const permission = await hasCameraPermission();
      setHasPermission(permission);
    })();
  }, []);

  const toggleCamera = () => {
    setType((current) =>
      current === CameraType.back ? CameraType.front : CameraType.back
    );
  };

  const snapAndSave = async () => {
    if (hasPermission) {
      let asset = await snapAndSavePhoto(camera);
      setUri(asset.localUri);
    }
  };

  return (
    <CameraContext.Provider
      value={{ type, uri, setUri, setCamera, toggleCamera, snapAndSave }}
    >
      {children}
    </CameraContext.Provider>
  );
};
```

Listing 18: cameraContext.js

The code snippet below (see Listing 19) illustrates how the *<Camera />* component is implemented in the *Camera.js* page (for the purpose of this example, I have omitted certain parts of the code, such as imports, styling, and inner components). It can be noticed that the

<*Camera />* component is rendered conditionally based on the value of the *isFocused* variable. This approach helps to follow the Expo Camera documentation (2024) recommendation, which suggests that the Camera should be active only when the screen is focused. Whenever users navigate to another screen, the value of the *isFocused* variable turns to false, and the Camera is unmounted.

```
export const CameraView = ({ navigation }) => {
  const isAndroid = Platform.OS === "android";
  const isFocused = useIsFocused();

  const { type, uri, setCamera, toggleCamera, snapAndSave } =
    useContext(CameraContext);

{...}

  return (
    <>
      {isFocused && (
        <Camera
          style={[mainContainerStyle, styles.container]}
          ref={(ref) => setCamera(ref)}
          type={type}
          useCamera2Api={isAndroid}
          ratio="1:2"
        >
            {...}
        </Camera>
      )}
    </>
  );
};
```

Listing 19: Camera.js

Following common standards of camera screen layouts, the most important clickable icons and a button that controls camera functionality are positioned at the bottom of the <*Camera/*> component and aligned horizontally. The <*Ionicons name="close-circle-outline"/*> component enables users to exit the Camera page and return to the *AddLocation.js* page. The <*CustomRoundButton/*> takes care of taking a photo by executing the *clickTakeAPhoto* function. If an image is taken successfully, it saves it and updates the *uri* state. The icon <*Ionicons name="close-circle-outline"/*> allows to toggle between the front and rear camera of a smartphone. (See Listing 20)

```
<View style={styles.buttonWrapper}>
<Ionicons
  onPress={clickNavigateBack}
  name="close-circle-outline"
  size={40}
  color="white"
```

```
/>
<CustomRoundButton onPress={clickTakeAPhoto} />
<Ionicons
  onPress={toggleCamera}
  name="sync"
  size={40}
  color="white"
/>
</View>
```

Listing 20: buttons in the Camera.js

4.6.3 Mikkel

Location view As my first task, I was responsible for implementing a detailed view for each location, so when the user press a location he can read about it and make it a favourite oasis (see Figure 6). The first step in the process was to create the frame since the basic react-navigation was under development. Therefore I started by hard coding the information to be able to style the basics properly. However, the location details were created as a component and prepared for merging with existing code, so it was easier to implement when the navigation was done.

In the *LocationDetails* component there are some codes related to the functionality for getting directions. Since I did not create this code it is excluded from the Listing (see Listing 21). Firstly, I had to decide whether I wanted to use conditional rendering to render the content on top of the current screen or use nested navigation. Since I found the experience more smooth with nested navigation, this was the solution I ended up with. Therefore, I used navigation as a prop so I could *navigation.navigate()* when a marker is pressed inside the *Map* component. One of the main challenges was getting the data from the *CustomMarker* component in the *Map* component. Usually, the props are used the other way around, but in this case, the data needed to be fetched from the *CustomMarker* component where it is called. I used the *useRoute()* hook, a component of React Navigation, to access the route's information and extract it to the *location* parameter. This was done by saving the data in the location that was extracted using *route.params*.

Lastly, we wanted the user to be able to toggle a favourite button to pick an oasis as a favourite or remove it again. For that reason, I used simple *useState* called *setFavorite* and then created a function *setFavorite(!favorite);* that always changes to the opposite state when called.

```

  function ViewLocation({ navigation }) {

  {...}

  const route = useRoute();
  const { location } = route.params;

  const [imageURL, setImageURL] = useState(null);
  const [favorite, setFavorite] = useState(false);

  useEffect(() => {
    setImageURL(
      "https://images.nationalgeographic.org/image/upload/
   t_edhub_resource_key_image_large/v1652303287/EducationHub/photos/
   earth-day.jpg"
    );
  }, []);

const changeFavorite = () => {
    setFavorite(!favorite);
  };

  return (...);
  }
```

Listing 21: LocationDetails.js from the *application* module

In the following code from the *LocationDetails* component the interface is presented and the structure of the visual content of the component (see Listing 22). The component is structured with an *outerContainer* and three containers inside the hierarchy: *imgContainer*, *detailsContainer* and *buttonContainer*. The image container needed to be styled using *relative* so the favourite button could be positioned on top of the image using *absolute*.

In the image container, a few things are worth noting. First of all the image source uses a conditional statement, so if an image is uploaded in the *Camera* component, this will be presented. If *location.uri* is falsy it defaults the image saved in *imageURL* in the context of *const [imageURL , setImageURL] = useState (null);* from the code in Listing 21. The image is fetched by using the *location.uri* as mentioned earlier. Next, the favourite button is implemented using a *Pressable* from RN. By using the onPress to call the *changeFavorite* function, the image will change depending on the state. If *favorite* is true the image with a filled icon is displayed and if it is false an icon with no fill is displayed.

In the next container, all information content is displayed. this is also fetched using *route.params* and added in *location.title* and *location.description*. Lastly, the button container is holding the "TakeMeThere!" button with location data for directions.

```
<View style={styles.outerContainer}>
      <View style={styles.container}>
        <View style={styles.imgContainer}>
          <Image
```

```
            source={location.uri ? { uri: location.uri } : { uri: imageURL }}
            style={styles.image}
          />
          <Pressable onPress={changeFavorite} style={styles.favContainer}>
            <Image
              source={
                favorite
                  ? require("../../assets/images/fav_filled.png")
                  : require("../../assets/images/fav_noFill.png")
              }
              style={styles.fav}
            />
          </Pressable>
        </View>

        <View style={styles.detailsContainer}>
          <Pressable style={styles.closeIcon} onPress={clickNavigateBack}>
            <Ionicons name="arrow-back" size={24} color="black" />
          </Pressable>
          <View style={styles.detailsTitleContainer}>
            <Text style={styles.detailsTitle}>{location.title}</Text>
          </View>
          <Text style={styles.detailsDescription}>{location.description}</Text>
        </View>
        <View style={styles.buttonContainer}>
          <Buttons
            title={"Take me there"}
            onPress={() => {
              handleTakeMeThere({ location: location.location });

              //set timeout allowing directions to be updated
              // before navigating back to map...

              setTimeout(() => navigation.goBack(), 200);
            }}
          />
        </View>
      </View>
    </View>
  );
}
```

Listing 22: LocationDetails.js from the *application* module

Recently visited component I was also in charge of creating the page for the recently visited page. The page was created with placeholders since other important tasks were not done at the time. For that reason, we did not have time to fully connect it to the existing functionality thus allowing the page to be populated either from geofencing or when the user has asked for directions. Nevertheless, the page was also prepared for merging at a later point.

The page was based on the component called *LocationContainer* (see Listing 23), which holds all the data of the component. This includes the title, image and the description, same as the aforementioned component. Looking back it would have been appropriate to only have one locationDetails component. The component presents information about a specific location within a visually cohesive card-like structure. This component takes in three essential props: title, img, and description, allowing us to dynamically populate the card with specific location details.

```
const LocationContainer = ({ title, img, description }) => (
  <View style={styles.card}>
    <View style={styles.cardContent}>
      <View style={styles.imageContainer}>
        <Image source={{ uri: img }} style={styles.image} />
      </View>
      <View style={styles.rightContainer}>
        <View style={styles.textContainer}>
          <Text style={styles.title}>{title}</Text>
          <Text style={styles.description}>{description}</Text>
        </View>
        <View style={styles.buttonContainer}>
          <TouchableOpacity
            style={styles.button}
            onPress={() => Alert.alert("hi")}
          >
            <Text style={styles.btnText}>Take me there!</Text>
            <Entypo name="location-pin" size={16} color="white" />
          </TouchableOpacity>
        </View>
      </View>
    </View>
  </View>
);
```

Listing 23: LocationContainer.js from the *components* module

The component also consists of a few containers for structuring the visual presentation of the data inside the component. Firstly, the outer container (<*View style* = *styles.card*>) defines the overall styling of the card. The image container (<*View style* = *styles.imageContainer*>) encapsulates the <*Image*> component from RN, responsible for displaying the location's image. The image source is dynamically set using the *img* prop, allowing us to add images for different locations. The important part is the hierarchy of the View component. When navigating to the inner containers: the right container (<*View style=styles.rightContainer*>) and the button container (<*View style=styles.buttonContainer*>), it is apparent that these include all data placeholders that will be updated with the props. They are structured in this way due to design reasons and the connected data. This is then used in the *LocationDetails* component.

For the recently visited page (see Listing 24) I created a useState *const [locationData, setLocationData] = useState([])* holding an array in *locationData* with the relevant data like id, title, image, and description. This was done to be able to dynamically change it along the way but also for debugging when working with the implementation. Likewise, the *addNewLocation* function is used for debugging and presenting new add locations to the array. It was important for us that a newly added location was added on top of the list which means that we need to added it to the beginning of the array and not the end. Therefore we simply used *setLocationData([newLocation, ...locationData]);*.

The *LocationContainer* component was then called inside a function called *const renderLocation*, which was responsible for rendering each location with all its data. The function uses an item as a parameter to extract the data from each object. The *id* is not represented here since we wanted it to auto-increment and thus always be unique

```
export const RecentlyVisited = () => {
  const [locationData, setLocationData] = useState([
    {
      id: "1",
      title: "Location 1",
      img: "https://images.nationalgeographic.org/image/upload/
    t_edhub_resource_key_image_large/v1652303287/EducationHub/photos/
    earth-day.jpg",
      description:
        "Lorem Ipsum is simply dummy text of the printing and
    typesetting industry. Lorem Ipsum has been the industry's standard
     dummy text ever since the 1500s, when an unknown printer took a
    galley of type and scrambled it to make a type specimen book.",
    },
  ]);

  const renderLocation = ({ item }) => (
    <View style={styles.item}>
      <LocationContainer
        title={item.title}
        img={item.img}
        description={item.description}
        style={styles.locationCard}
      />
    </View>
  );

  const addNewLocation = () => {
    const newLocation = {
      id: String(locationData.length + 1),
      title: ‘Location ${locationData.length + 1}‘,
      img: "https://images.nationalgeographic.org/image/upload/
    t_edhub_resource_key_image_large/v1652303287/EducationHub/photos/
    earth-day.jpg",
      description:
        "Lorem Ipsum is simply dummy text of the printing and
    typesetting industry. Lorem Ipsum has been the industry's standard
     dummy text ever since the 1500s, when an unknown printer took a
    galley of type and scrambled it to make a type specimen book.",
    };
    setLocationData([newLocation, ...locationData]);
  };

```

Listing 24: RecentlyVisited.js from the *application* module

For the recently visited page, we wanted locations to be displayed dynamically on a list that the user could scroll through and see. Therefore, we used a *Flatlist* from RN (see Listing 25. The *Flatlist* is wrapped in a *listContainer* to properly style and structure the list in relation to the rest of the page. The list is using the *data* prop to extract *locationData* from the previous useState. This was originally supposed to be data from the actual locations, but at this point only acting as a placeholder. We also needed to extract the key to keep track of each object and then use the *renderItem* prop to call the *renderLocation* function responsible for rendering each item from the array.

```
<SafeAreaView style={styles.container}>
      <StatusBar style="auto" />
      <Text style={styles.header}>Recently Visited</Text>

      <View style={styles.listContainer}>
        <FlatList
          data={locationData}
          keyExtractor={(item) => item.id}
          renderItem={renderLocation}
          contentContainerStyle={styles.flatListContainer}
        />
      </View>
    </SafeAreaView>

```

Listing 25: recentlyVisited.js from the *application* module

Navigation - location view Since I decided to use nested navigation for accessing the details for each location it had to be merged with the existing app navigation (see Listing 26.

However, this was later changed as mentioned in section 4.6.2 to include the functionalities of adding location and accessing the camera.

To achieve this, I used a conditional rendering by checking if the value of the *isAuthenticated* variable. If it's true, it renders the *<LocationDetailNav />* component; otherwise, it renders the *<AuthenticationNavigation />* component. This ensures that the user is only navigating to the *LocationDetailNav* if he has succesfullt logged in.

```
<NavigationContainer>
      {isAuthenticated ? <LocationDetailNav /> : <
    AuthenticationNavigation />}
    </NavigationContainer>

```

Listing 26: index.js from the *navigation* module

Inside *LocationDetailNav* is a nested navigation using a *Stack Navigator*. Inside this component from RN two screens are represented: *ApplicationNavigation* and *LocationDetails*. The *ApplicationNavigation* holds the navigation of the entire application including a bottom navigation, whereas the *LocationDetails* holds the screen for navigating to the details screen for each location.

```
    <Stack.Navigator>
      <Stack.Screen
        name="App"
        component={ApplicationNavigation}
        options={createScreenOptionsForApp}
      />
      <Stack.Screen
        name="LocationDetails"
        component={LocationDetails}
        options={createScreenOptionsForApp}
      />
    </Stack.Navigator>
  \begin{lstlisting}[ caption=recentlyVisited2.js from the \textit{
    components} module, label=lst:recentlyVisited2]

```

Listing 27: LocationDetailNav.js from the *navigation* module

Figma prototype In the brainstorming process we had three main ideas as mentioned in subsection 4.1. For my part, i was in charge of creating an initial representation of the Scavenger Hunt idea in Figma (see Figure 17). The prototype was created with basic transitioning and placeholder items to visualize how the app could look. I explored having two login types: teachers and students, so teachers could create scavenger hunts and students could participate in them.

5 Discussion

In this section we will discuss our main challenges and how these were solved. We will discuss our process and what could have been done differently. We will address our final result and whether our goal was reached. Finally, we will discuss future work for our app that could be beneficial to implement.

5.1 Main challenges

One of our biggest challenges was planning out and working simultaneously on components building on top of other components that were under development. This meant that some tasks had to be postponed to a later point even though they were crucial and time-consuming. We were somewhat aware of this before starting our development, so in our planning, we tried to accommodate it by creating our milestones and deadlines. Additionally, we delegated smaller tasks while bigger ones were under development. We also decided members who had to wait for a task should research and figure out their technical approach. That way he could explore the coding task at hand and even try it out in a separate Git branch. As a result, the tasks could be completed much quicker.

Another challenge we experienced was merging branches in Git. We sometimes experienced errors after merging code and had trouble figuring out why they occurred. We tried out multiple approaches such as changing some code, researching, and looking into our package.json file. In the end, we figured out that the solution was to run ”npm install” in the terminal and the issues were fixed. This helped us with many conflicts since we often imported new packages for our application.

When implementing markers on the map we experienced a bug that we could not fix within the time frame. This was a major challenge that took some time to understand. Whenever a user adds a location, a bug appears if he chooses to add the location to his current location. The reason is when I navigate back to the Explore screen, it seems like the location has not been added. However, this is not the case. The location is successfully added but it is hidden behind the marker for the current location. In the end, we did not have time to fix this bug.

Last but not least we wanted to have a smooth transition from the map to the Location Details screen. As a first approach, we explored using conditional rendering and rendering the screen on top of the rest of the content. This gave us some issues regarding the styling of the rest of the screen and the fact that we wanted the user to be able to navigate to the screen from multiple places e.g. from the Recently Visited page and the Explore page. Therefore we decided to use nested navigation, which was quite complicated. The navigation had to be modified multiple times since we needed nesting on multiple occasions such as the camera component. In consequence, we had to go through some redundant tasks multiple times.

5.2 Final result

Overall our milestones were successfully reached and our goals for the prioritized features were completed in time. Our planning was one of our strongest sides during the development and was carefully followed throughout the course. Therefore, we are very satisfied with the final result of the app. However, we wanted to have some more feedback in our login screen and other places in the app. Likewise, we wanted to incorporate the favourites to the account page, which we also prepared for on the Location Details screen. Here we present the user with a ”star” button that can be toggled to simulate that it is added to the favourites. This feature was planned as a ”nice to have” from the beginning, but we wanted to implement it to create a more coherent app. Unfortunately, other parts like getting directions and adding camera support were time-consuming and for that reason, we did not have enough time.

5.3 Future work

We have some tasks that need to be done to create a more fluid experience and a more functional application. First of all, future works of the app would include moving from async storage to a regular database for locations in the app to avoid saving data locally on the device. Additionally, we would like more unified style modules, so modifications in the app styling would be less painful and more efficient. Regarding geolocation, we would also have liked to make use of geofencing when adding locations to the Recently visited page. At the moment, locations are added when a user presses the ”TakeMeThere!” button and asks for directions. Instead, we would like to use geofencing to mark locations as recently visited once the user has been physically in that area. As mentioned earlier, the favourites feature needs to be implemented for future works, so the user can mark the locations he likes the most and make them stand out. That way we can significantly improve his experience by reducing the time for him to perform searching tasks.

Bibliography

Brauer, B., Ebermann, C., Hildebrandt, B., Remané, G., and Kolbe, L. M. (2016). Green by app: The contribution of mobile applications to environmental sustainability. *Journal Name*, Volume Number(Issue Number):Page Range.

Expo (2024). Expo camera documentation. Online documentation.

Google (2024). Firebase authentication documentation. Online documentation.

Google Maps Platform (2024). Google maps directions api documentation. Accessed on: 20 februari 2025.

Hansen-Møller, J., Konijnendijk, C., and Caspersen, O. (2011). Betydningen af storbyens rekreative områder for storbybefolkningens sundhed og velvære, tryghedsfølelse, trivsel og glæde samt de økonomiske konsekvenser for borger og samfund. Technical report, Arbejdsrapport nr. 136, Skov & Landskab, Københavns Universitet, Frederiksberg.

Lupanda, I. S. and Rensburg, J. T. (2021). Design guidelines for mobile applications. *Proceedings of the 15th International Conference on Interfaces and Human Computer Interaction 2021 and 14th International Conference on Game and Entertainment Technologies 2021.*

Narcizo, F. (2023a). 05-1_appnavigator. GitHub repository.

Narcizo, F. (2023b). 05-2_firebaseauthentication. GitHub repository.

Narcizo, F. (2023c). 10-1_cameraphoto. GitHub repository.

park4night (n.d.). Home. Accessed: 20 februari 2025.

Visit Denmark (2022). Turismen i danmark. København.