# FUN!GUS

## PROGRAMMING MOBILE APPLICATIONS

COURSE CODE | KAPRMOA1KU

**GROUP 4 |**
PARNUNA BRØNLUND (parb@itu.dk)
VICTOR ROSAGER CHRISTENSEN (vchr@itu.dk)
JOSEPH AUGUSTIN BYRNE (joby@itu.dk)
NINA FLERON LETOFT (nifl@itu.dk)

IT-UNIVERSITY

EXPO LINK

**PRESS HERE**

THE PROJECT
RUNS ON **SDK 53**

# TABLE OF CONTENTS

# 1 | INTRODUCTION

During our initial brainstorming sessions, we explored various project concepts including water reservoir mapping, crowd-sourced bookshop locations, and ultimately settled on our mushroom cultivation application. This project emerged from recognizing Copenhagen's vibrant coffee culture and the untapped potential of coffee grounds as a sustainable resource for urban mushroom growing.

FUN!gus addresses the growing need for sustainable practices in urban environments by creating a community driven platform that transforms waste into valuable resources. Our motivation stems from the observation that coffee grounds, despite their nutrient rich properties, are often discarded when they could serve as excellent growing medium for mushrooms. The application facilitates local exchanges between users seeking to donate coffee grounds, receive them for cultivation, or trade homegrown mushrooms and spores.

We chose to develop our mobile application iOS devices to ensure a polished, consistent user experience that aligns with our target demographic in Copenhagen. The iOS ecosystem's design principles complement our focus on a clean and intuitive interfaces that encourage sustainable behavior through seamless interaction.

Collaborative development proved essential for this project's success. Working as a team of four allowed us to distribute expertise across different components, from UI design and navigation systems to camera integration and map functionality. Group collaboration fostered creative problem solving, peer review, and knowledge sharing that elevated our final product beyond what any individual could achieve alone. This collective approach mirrors the community driven ethos that FUN!gus itself promotes.

# 2 | CONCEPT – FUN!gus

Expo Link: Press here

Our concept is an application called FUN!gus, which we developed to support sustainable mushroom cultivation by enabling users to grow mushrooms using recycled coffee grounds. FUN!gus facilitates local resource exchange by connecting users who want to exchange mushrooms, donate used coffee grounds, or receive coffee grounds for cultivation purposes. The overall goal is to promote environmentally conscious behavior through community sharing practices.

When the user logs in to the application (Figure 1), they are directed to the HomeScreen (Figure 2), where they can upload their profile photo or define their purpose. The user can choose one or more of the following purposes: exchanging mushrooms, donating coffee grounds, or

receiving coffee grounds. These options are supposed to set up the user's profile. Other user profiles are displayed on the Map screen (Figure 3), where each user is displayed as a marker with their chosen interest. The user can then press the marker on the MapScreen, and a popup is displayed, with the option to start a direct conversation with the other user and is redirected to the MessageScreen (Figure 4). On the MessageScreen, the user can filter their conversations according to the selected interest and engage in a chat to arrange exchanges (Figure 5). In addition to the exchange functionalities, the application provides informational content on the FeedScreen (Figure 6). The screen consists of four cards: a mushroom recipe, a frequently asked questions section, information regarding the concept's sustainability impact, and general information about the three main purposes of the application.

A typical user scenario might involve a user who collects coffee grounds at their workplace and wishes to donate them. After selecting “donate coffee grounds” on the HomeScreen, the user is presented with a nearby post from others who are looking to receive coffee grounds. Upon selecting a relevant post on the MapScreen, the users initiate a conversation to arrange a pickup.

FUN!gus addresses two of the United Nations Sustainable Development Goals. First, Goal 12: Responsible Consumption and Production, by supporting the reuse of organic waste and promoting circular food systems. Second, Goal 11: Sustainable Cities and Communities, by encouraging local engagement, reducing waste, and contributing to more sustainable ecosystems. Initial research indicates that coffee grounds are an underutilized yet nutrient-rich resource, often wasted despite the potential value. By creating an easy-to-use application for exchanging resources, FUN!gus empowers individuals to actively participate in sustainable practices in everyday life.


Goal 11 | Make cities and human settlements inclusive, safe, resilient, and sustainable


Goal 12: | Ensure sustainable consumption and production patterns

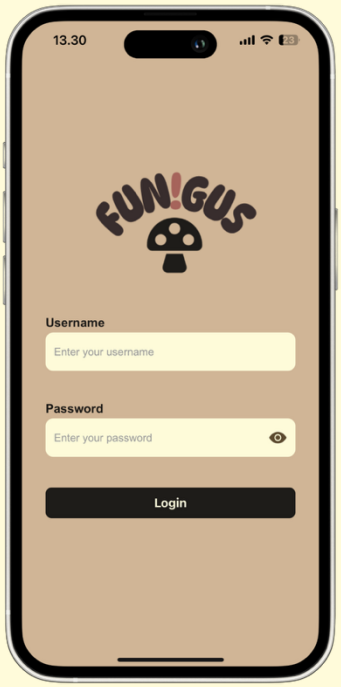


Figure 1 | LoginScreen


Figure 2 | HomeScreen

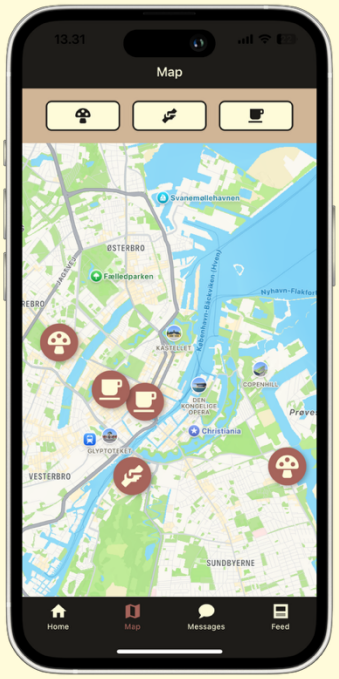


Figure 3 | MapScreen

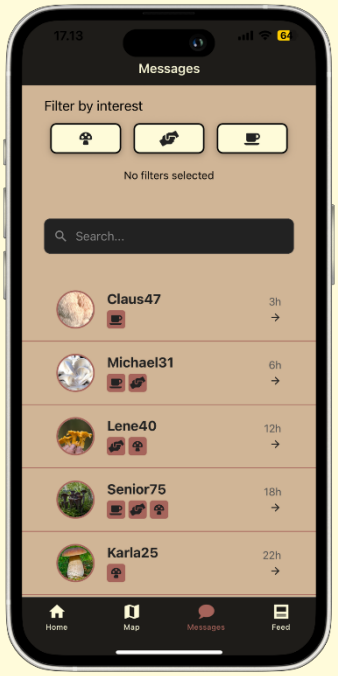


Figure 4 | MessageScreen


Figure 5 | ChatScreen


Figure 6 | FeedScreen

# 3 | BACKGROUND & RELATED WORK

We got inspired by a business called ‘Beyond Coffee’ (beyondcoffee.dk, 2025). They sell ‘grow boxes’ with mushroom spores or coffee grounds, easy to grow at home in your kitchen.

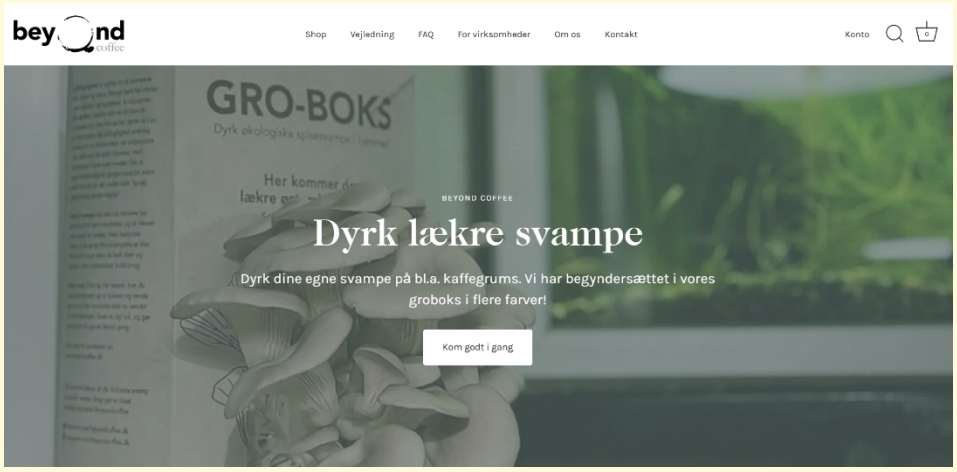


Figure 7 | Frontpage – Beyond Coffee

The concept was cool, and since Copenhagen is a coffee city and most people live in apartments, we thought we could do something similar yet sustainable and without money involvement. The website has a clean and user-friendly interface with a calm design and a Nordic appeal.

When it comes to mushroom related applications, the concepts mostly revolve around how to find mushrooms in nature or how to reduce carbon foodprint by buying local crops. With our idea, we saw the potential of combining both, however designed for people living in the city. Our application development was inspired by combining the best design elements from popular social media platforms to create an intuitive user experience.

We were inspired by Snapchat's camera interface design. The camera component features a circular capture button that mimics Snapchat's iconic ‘dual ring’ aesthetic. We implemented camera flip functionality and a minimalist full-screen interface that keeps the camera clean and user-friendly.

In our ChatScreen, we drew inspiration from Facebook Messenger's interface design. The component features a user header, scrollable message list, and bottom-positioned message input that mirrors Messenger's familiar design. We wanted to add message deletion with confirmation alerts and timestamp functionality for a complete message experience, providing intuitive conversation flow with clear visual hierarchy.

# 4 | APPLICATION DEVELOPMENT

This section documents our development process for FUN!gus, from initial Figma mockups to working prototype. We explored our collaborative workflow through GitHub, component architecture decisions, and code structure choices that enabled effective React Native development. The section covers integration of compulsory features, geolocation, camera, and map functionality, alongside our navigation system. We also examine peer feedback integration and mobile design principles that guide our user experience decisions.

## 4.1 | MOCKUPS IN FIGMA

At the beginning of our process, we began brainstorming which screens were essential for solving the main tasks for the course. We therefore decided that each of us would individually present application ideas to the group, drawing inspiration from the UN goals. Nina's idea pitch to the group was voted the best, so we decided to create an application for exchanging mushrooms and coffee grounds.

From generating initial ideas (Figure 8), we began establishing our visual direction. Figma has been central to our workflow throughout the entire design process (see Figma link here).

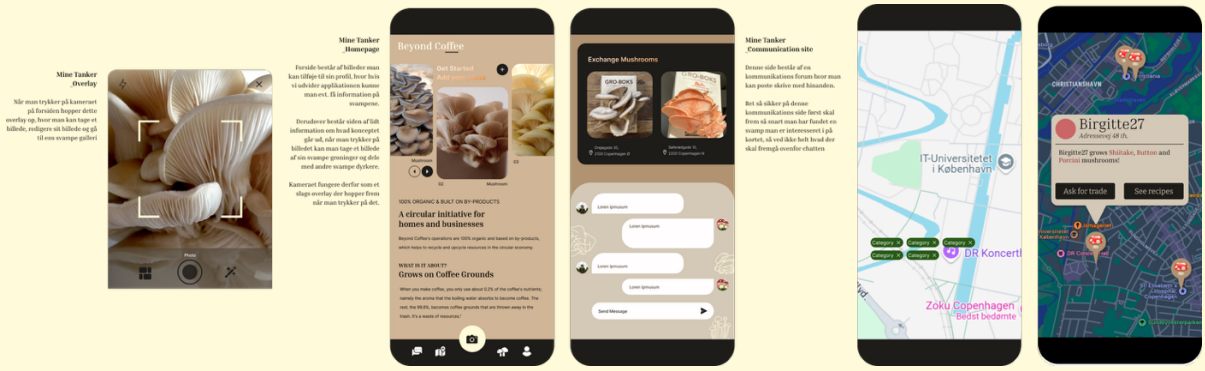


Figure 8 | Brainstorming & early sketches in Figma

Choosing the right colour palette was one of our first decisions. We wanted warm, earthy tones that would convey the natural, sustainable feel of mushrooms and coffee grounds. We decided on these Hex code colours: dark brown/black: 1E1C19, light Brown: D0B596, beige: FEFBD9, and coral: A6645B, (Figure 9). The palette is efficient to use as contrasting colours.

Figure 9 | Our chosen colour combination

Our button designs evolved to match this aesthetic - clean and functional, but with subtle details that give them character and that vintage touch we were aiming for (Figure 10).


Figure 10 | Our chosen colour combination

The fonts and overall interface style followed naturally from these foundations. We found ourselves very much aligned with our design perspective and decisions; we all wanted a clean, user-friendly interface with a vintage appeal.

The process of developing our wireframes helped us to align our ideas. The HomeScreen underwent a transformation, from simple and not that intuitive, to an updated version with additional information, a functional camera, and a slider with directions to the MapScreen (Figure 12).

The RecipeScreen also evolved from “just” a RecipeScreen to a multipurpose FeedScreen with both a mushroom recipe (RecipeOverlay), an FAQ page (FAQOverlay), a page regarding the concept (CoffeeOverlay), and page with information's about how our concept can help with lowing the sustainability impact (Sustainable Overlay), see the four informational cards on (Figure 14).


Figure 11 | Initial ProfileScreen

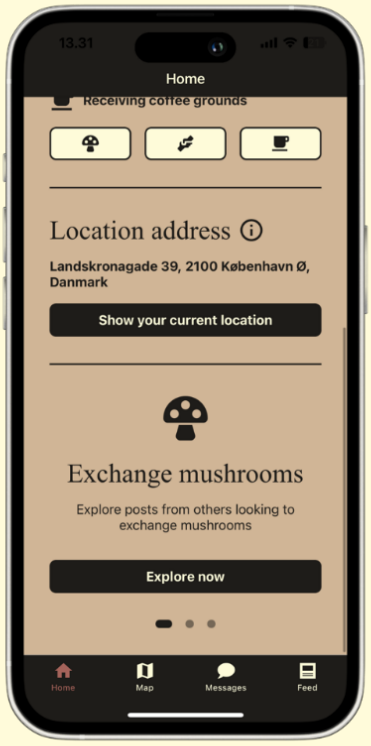


Figure 12 | Updated HomeScreen


Figure 13 | Initial RecipeScreen


Figure 14 | Updated FeedScreen

## 4.2 | GUIDELINES FOR MOBILE APPLICATIONS

We incorporated established design principles for mobile user experience (UX) and user interface (UI) as outlined by Lupanda and Janse van Rensburg (2021), when developing our application, FUN!gus. A key focus in our design was to reduce mental effort for users and make the application easy to use in different user situations. We followed the principle of *cutting out clutter (M1)* by maintaining a minimalistic interface with only essential content shown on each screen. This would possibly help our users stay focused and reduce the risk of confusion. Similarly, navigation and interactive elements were kept clear and simple to guide the user's experience and their actions effectively. This would help to support intuitive mappings between *controls and their effects (H5)*.

To support accessibility and easy interaction, we designed *finger-friendly tap targets (M4)* with sufficient size and spacing, which is especially important for our users when navigating quickly or in motion both vertically & horizontally and between the screens. Additionally, elements were made *clearly visible (M6)*, using high contrast colors and accessible fonts to enhance readability for users with low vision or color blindness. This also aligns with the HCI principles by offering *informative feedback (H2)*, ensuring that users can easily see what is happening on the application and what options are available to them.

We recognized that users often access applications in shifting modes, when they are either *bored, busy, or lost (M11)*. Our design accommodates these mobile principles through quick interactions, clear navigation, and immediate feedback. An example of this is our alert popup, when attempting to delete a message. When the trashcan icon is pressed, a popup *confirmation is triggered (M16)*. This gives the user both a reassuring explanation and an option to cancel the action. This implements both *informative feedback (H2)*, *simple error handling (H4), and dialog closure (H9)*, while keeping language and visual design user-friendly.

By applying these design principles, consequently throughout our interface, we ensure that FUN!gus remains user-friendly, responsive, and accessible for real-world use.

## 4.3 | COLLABORATION

From start to finish collaboration has followed a similar structure: weekly meetings would serve to identify the next step in development and individual tasks would be delegated to each group member. In the design stage, this would include designing individual Figma screens or specific application flows. Each member's work would then be evaluated with assistance provided if needed.

For the development of the application, each group member assigned themselves one or two screens depending on the complexity. The assigned member would then have responsibility for the functioning and ensuring that the screen/screens matched the design mockup in our collaborative Figma document. The development of the screens was managed individually with weekly meetings to ensure congruency, evaluation, and assistance where needed.

The purpose of collaborative development in GitHub repository was setting up a branch for each member within the group as well as a main branch. Each member would develop their screens in their own branches. Only after the individually developed code had been presented to the rest of the group, first then the individual branch was merged into the main branch. Working collaboratively in GitHub was a new experience for all of us; however, after a few merges, it became completely integrated into our development process throughout the course.

## 4.4 | IDENTIFYING COMPONENTS

From the beginning of the design process, and continually afterwards, we have been cognizant of breaking down our application into components. As custom and reusable components were presented as essential for the project, we were alert when it came to the identification of which elements should be structured as custom components.

After developing the wireframes in Figma, we decided to create a note below each, to make sure each screen had enough components and that scalability would be possible, if changes were to happen during programming. A screenshot of the labels added can be seen below in Figure 15. Each label was named after what we had identified could be a custom component in the application. As our final prototype largely matches the concept conceived in Figma, many of the components identified in the design process have also stayed during development.


Figure 15 | Mockup – initial component list

A section of components that have been essential throughout development have been standardized text and button components. Rather than ensuring that the styling on each screen or custom component follows the same design, we developed a custom button with label text, header text and body text that would be called whenever needed. As such, all buttons and text throughout the application are congruent.

Almost every component identified during our design phase in Figma has been included in the application. However, through our design process we chose to build additional design features and include additional components which are not presented in Figure 15.

## 4.5 | CODE STRUCTURE

Our application is built using React Native as a prototyping tool. We chose this framework because it allowed us to create a mobile-friendly interface and test the functionality quickly. The application architecture is structured stringently into multiple folders to keep the code organized and easy to navigate through. The components folder contains reusable UI elements such as buttons, text styles, overlays, and input fields. The screens folder contains the main views of the application like login, home, map, message, chat, and feed. Each screen is responsible for displaying relevant components and managing the layout of a specific part of the user experience.

To separate logic for presentation and enhance modularity, we created a hook folder. Hooks in React are functions that allow us to use state and features in functional components. For example, we use a state hook like user-filtering and a custom hook named use-interest-icons, to manage filtering logic and visual preferences across different parts of the application. This approach helps to increase reuse of code and keep our logic consistent.

Static data used through the application, such as predefined filters, user profiles, and map locations, is placed in the data folder. The constants folder holds global values including fonts, colors, and other styling settings, which makes it easier to ensure visual consistency and make changes when needed. Application navigation is managed in the navigation folder, which contains the logic for both tab and stack navigators.

Each file is commented to clarify the purpose and functionality of the code. This improves collaboration and ensures the application is easier to maintain and scale. Overall, the folder structure, use of comments, and folder design reflect our focus on clarity, re-usability, and efficiency.

## 4.6 | DECISIONS

When we began to program FUN!gus, we decided to create a functional prototype that mimics the user-flow of the intended application.

For that reason, we have made the following decisions when programming FUN!gus:

- We decided to implement all the screens of the intended application to simulate the user-flow of the intended FUN!gus application, only the RegisterScreen is missing.
- We decided to use hardcoded arrays of data to mimic a database to save time.
- We decided to create a profile context provider using the useContext hook from the React framework to mimic the backend and database functionality of the profile settings, thus saving time.
- We decided to simulate a real authentication flow using the useState hook from the React framework to save time.
- We decided not to implement the functionality to make the FUN!gus application work for multiple users because it would take too long to make.
- We decided not to implement geolocation on screens other than the home screen in FUN!gus to save time. This resulted in us only implementing geolocation on the home screen via the geolocation location component and implementation of hardcoded location data on the other screens in FUN!gus, to simulate the user-flow of the intended application.
- We decided to make the button of the map popup navigate to the message screen instead of a matching chat screen in FUN!gus, because it was too complicated for us to develop.
- We decided to hardcode the map makers of the users' posts to have only one icon based on a hardcoded main interest value. This decision was made because we found it too complicated and time-consuming to implement a system that assesses the user's multiple purpose interests.

## 4.7 | REQUIRED COMPONENTS & NAVIGATION

Certain components were compulsory inclusions in the application and were naturally included in the final prototype. These components include geolocation, a camera, and a map. Due to the complexity required of a prototype like this, navigation has also become compulsory. While the inclusion of compulsory components can potentially be restrictive, they were in our case points of departure for our creative process and helped us reach our final design vision with relative ease.

### 4.7.1 | GEOLOCATION & GEOFENCING COMPONET

The geolocation component is set up in the HomeScreen in FUN!gus and is designed to make other users see the user's post on the map which is located on the user's current location. The

post of the users shown on the MapScreen is defined by the purposes the user has selected on the home screen.

It is possible for users to hide their current location whenever they want, so others can't see their post when the user is unavailable.

### 4.7.2 | CAMERA COMPONENT

The camera component is implemented in the CameraScreen in FUN!gus and is used to access the camera on the user's device and take a photo that can be used as the user's profile image. The camera component is also connected to a preview component that has several purposes in FUN!gus. The preview component is also implemented in the camera screen and is used to check the photo taken with the camera component, save the photo taken to the user's media library on their device, delete the taken photo to replace it with a new photo, or use the taken photo as the user's profile image.

### 4.7.3 | MAP COMPONENT

Central to FUN!gus is the implementation of a <mapview> to find other users for the exchange of mushrooms and coffee grounds. The map acts as a hub of users that are placed throughout Copenhagen through location-based map markers. Each of these markers acts as a gateway to conversation and subsequent mushroom-related activities. As the application promotes actual real-world interactions between human beings, having a map as the central user-hub allows the user to identify nearby users and minimize travel.

### 4.7.4 | NAVIGATION

The navigation of FUN!gus is built using some form of authentication and the native stack and bottom tabs navigation components from the React Navigation library.

FUN!gus consists of three layers of screens as you can see on the sitemap:

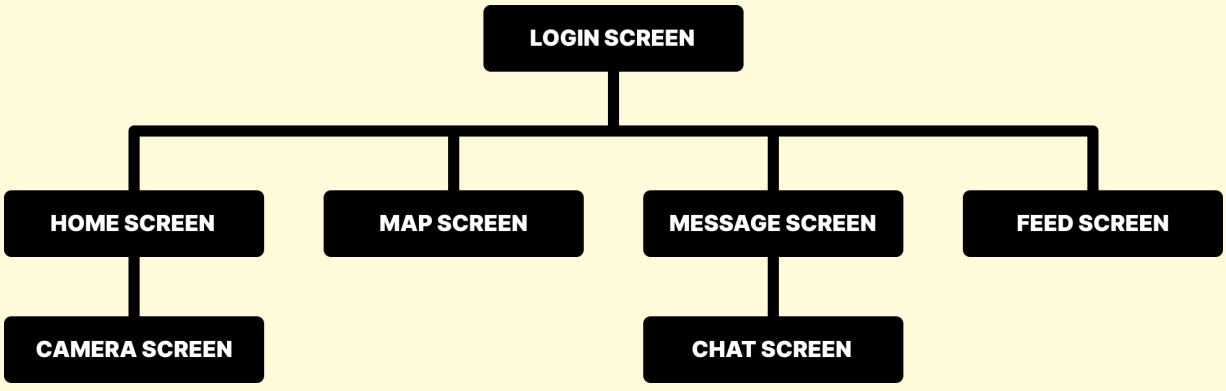


Figure 16 | Own creation - Sitemap

The first layer of screens in FUN!gus consists only of a LoginScreen. This screen is connected to the second layer of screens with some form of authentication, which is configured using the useState hook from the React framework.

The more general screens of FUN!gus is in the second layer of screens and consists of the following screens: The HomeScreen, The MapScreen, The MessageScreen, The FeedScreen

For the navigation between the general screens in the second layer, we used the bottom tabs navigation component from the React Navigation library, which allows navigation between the general screens via the accessible tab menu, that is automatically added to the mobile application when using the bottom tabs navigation component.

The third layer of screens consists of the camera screen and the ChatScreen, and to connect these screens with some of the screens in the second layer, we have nested the bottom tabs navigation component, which connects all the screens in the second layer, into a native stack navigation component from the React navigation library. This allows navigation from the HomeScreen to the CameraScreen and navigation from the MessageScreen to the ChatScreen in FUN!gus.

The navigation can also be viewed on Figure 17 below. The navigation is set up in our Figma file, which is linked in the beginning of section 4.1 (Mockups in Figma) or in the figure text below.

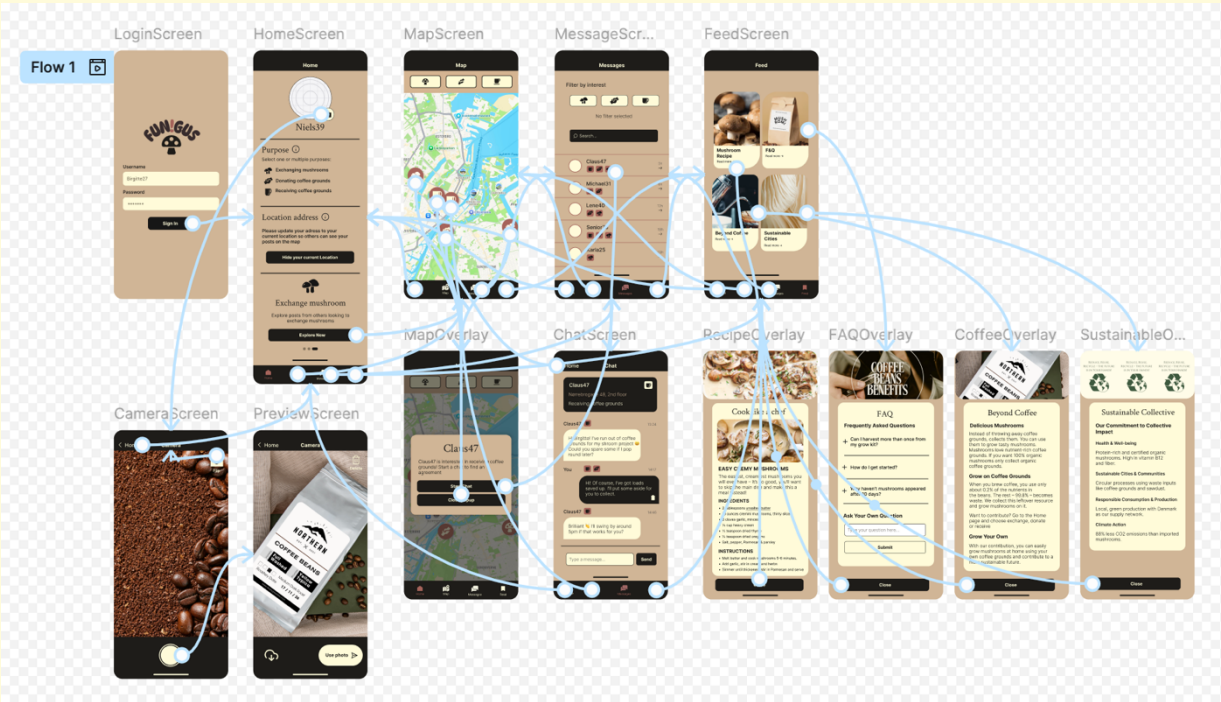


Figure 17 | Figma Prototype Navigation (Figma Link here)

## 4.8 | PEER-EVALUATION FEEDBACK

During the WIP prototype evaluation session, we received valuable feedback that helped us refine both the technical and conceptual direction of our application. One of the main takeaways was the need for clearer communication of password requirements during the login process. As a result, we decided to display the minimum character count more prominently, we also considered adding a logout feature to facilitate usability, however as this was not necessary for the application concept we decided to focus on the login screen instead. Additionally, we received a lot of feedback regarding the recipe screen. Our peer longed for a screen that was more about information about the concept. We therefore chose to develop an informational feed screen, with facts about mushrooms and coffee grounds, either as a login screen popup or a persistent component, to strengthen the thematic narrative. From a content perspective, we decided to transform the recipes screen into a broader feed combining a recipe, growing tips, a FAQ section, and information about sustainability, to enhance user engagement. Furthermore, our peer also longed for more mushroom-related humor and iconography to support a cohesive and playful interface. The feedback given was a great idea; however, if more humor needed to be incorporated it needed to be streamlined across the screens, and due to more relevant feedback, we decided to hold back on this matter. Technically, we restructured the code to improve readability by adding comments and separating components more logically. This decision was driven by feedback highlighting difficulties in understanding code dependencies and flow. That required some of us to refactor the code into separate files, which was a good learning experience and makes it easier to maintain when it is needed. Lastly, we aimed to complete the map screen and continue aligning the design with our core theme of sustainable mushroom cultivation through community engagement.

## 4.8 | BEST PRACTICES

We have chosen some best practices when programming, these include our conscious decision to prioritize clear naming throughout our React Native project. When we write functions with clear names like UserList instead of something vague like Peeps, everyone on the team knows what the function does without having to read through the actual code to figure it out. Using simple code principles means writing code that is easy to read and understand by choosing clear names. Additionally, we were committed to keeping each component focused on doing one thing to the fullest, rather than trying to make them handle multiple responsibilities.

One of our strongest agreements has been to eliminate repetition wherever possible. We have established shared components and helper functions so when we needed to change something later, we only had to update it in one place using the "Don't Repeat Yourself" (DRY) principle, which means write code once, use it multiple places, like our text and screenlayout components. This is also parental in our styling, here we chose to centralize our colors and configuration into dedicated files. Which has become our own design system within the code.

Consistency has become non-negotiable for us. We developed a pattern for file structure, naming, and component architecture, and kept to these throughout the project. It means anyone in our group can search in the codebase to lookup wished findings and see how they function. Furthermore, it has become evident that performance comes naturally when our components are focused and independent.

All of this has made our testing and error search much more straightforward. Since our components have clear responsibilities and do not depend heavily on each other. This fosters the possibility to test them in isolation and trust that our tests mean something or show where something is wrong.

# 5 | INDIVIDUAL CONTRIBUTIONS

## 5.1 | LOGIN SCREEN – NINA

The <*LoginScreen*> serves as the sign-in interface for our application. It allows the user to enter a username and password, validates the user's input directly within the application, and updates the signed-in state if the inputs are valid. If either input is invalid, an error message is displayed in the corresponding field. This validation is handled by the <*validateInputs*> *function* that returns a Boolean value.

The screen is structured within a <*KeyboardAvoidingView*> to prevent UI elements from being hidden by the on-screen keyboard, especially on iOS devices. A <*ScrollView*> enables vertical scrolling when the keyboard is active, ensuring all content remains accessible. To enhance usability, a <*Pressable*> component is included to dismiss the keyboard when tapping outside the input fields.

State is managed using the *useState* hook. Input values for username and password are stored in their respective state variables. Two additional state variables store error messages, while another controls whether the password is visible or hidden. This structure enables real-time validation and immediate visual feedback to the user.

The <LoginScreen> consists of three main components, including the <ScreenLayout>. The code snippet demonstrates how the screen is structured using *<KeyboardAvoidingView> and <Pressable> to manage the keyboard behavior. It also illustrates how <UsernameInput>, <PasswordInput>, and <IOSButton> are integrated. The use of props and state enables error handling and password visibility control.*

```
  <KeyboardAvoidingView
    style={styles.container}
    behavior={Platform.OS === "ios" ? "padding" : undefined}
  >
    {/* Dismiss keyboard if user taps outside input fields */}
    <Pressable style={styles.container} onPress={Keyboard.dismiss}>
      <View style={styles.innerContainer}>
        {/* App icon */}
        <View style={styles.logoWrapper}>
          <Image
            source={require("../assets/logo.png")}
            style={styles.image}
          />
        </View>

        {/* Username input */}
        <UsernameInput
          value={username}
          onChangeText={setUsername}
          error={usernameError}
        />

        {/* Password input */}
        <PasswordInput
          value={password}
          onChangeText={setPassword}
          error={passwordError}
          showPassword={showPassword}
          onTogglePassword={() => setShowPassword(!showPassword)}
        />

        {/* Login button */}
        <IOSButton title="Login" onPress={handleLogin} />
      </View>
    </Pressable>
  </KeyboardAvoidingView>
);
```

Figure 18 | LoginScreen Structure – Combining Components with Keyboard Handling

## 5.1.1 | USERNAME-INPUT COMPONENT

The <*UsernameInput*> component is a custom input field specifically designed for entering a username. It uses the reusable <*TextErrorInput*> component, which displays a label above the field, a placeholder inside the input, and an error message if validation fails. The validation

checks whether the username is at least four characters long. If the condition is not met, an error message appears below the input to guide the user.

The code snippet below shows how <UsernameInput> passes values and error handling to the <TextErrorInput> component, making the field both reusable and easy to manage.

```
export default function UsernameInput({ value, onChangeText, error }) {
  return (
    <TextErrorInput
      label="Username" // Label above input field
      placeholder="Enter your username" // Placeholder inside input
      value={value} // The current value
      onChangeText={onChangeText} // Updates the value on typing
      error={error} // Error message if validation fails
    />
  );
```

Figure 19 | UsernameInput Component – Passing Props to TextErrorInput

### 5.1.2 | PASSWORD-INPUT COMPONENT

The <*PasswordInput*> is a custom input field built using <*TextErrorInput*>. It includes a toggle icon that allows the user to show or hide the password text. The icon is controlled by the <*showPassword*> Boolean, which is toggled by the on *TogglePassword function*. The input is validated to ensure a minimum length of four characters, and an error message is displayed if the condition is not met.

The code below shows the implementation of the <*PasswordInput*> component, which passes props into <*TextErrorInput*> and conditionally renders an eye Icon inside the field. The icon allows the user to toggle password visibility, improving usability while maintaining security. By default, the password is hidden and is only shown when *showPassword* Boolean is *true*.

```
export default function PasswordInput({
  value,
  onChangeText,
  error,
  showPassword,
  onTogglePassword,
}) {
  return (
    <TextErrorInput
      label="Password"
      placeholder="Enter your password"
      value={value}
      onChangeText={onChangeText}
      error={error}
      secureTextEntry={!showPassword} // Hide text if not showing password
      rightIcon={
        <Ionicons
          name={showPassword ? "eye-off" : "eye"} // Eye icon toggles
          size={24}
          color="#5D4B30"
        />
      }
      onIconPress={onTogglePassword} // Handles toggle logic
    />
  );
```

Figure 20 | PasswordInput Component – Password Visibility Toggle

### 5.1.3 | TEXT-ERROR-INPUT COMPONENT

The <TextErrorInput> component is a reusable input field used for both the username and password inputs. It is designed to display a label above the input field, an optional icon (such as an eye icon for toggling password visibility), and an error message below the field when validation fails. The component receives all necessary props, including the input value, placeholder text, change handler, and error message. If an error is shown, a red border appears around the input field, and the error message is shown, a red border appears around the input field, and the error message is displayed in red text below. This provides immediate user feedback when the input is invalid.

Additionally, if a *right Icon* is passed, it is rendered inside the input field with a <TouchableOpacity>, making it interactive (e.g. for toggling password visibility).

The code snippet below highlights the main return block of the component, showing how it displays the label, input field, optional eye icon, and conditional error message. It also demonstrates how the component is styled to ensure visual consistency across different screens. This component plays a central role in maintaining a clean visual appearance throughout the LoginScreen, while also supporting accessibility and effective error handling.

```
export default function TextErrorInput({
  label,
  value,
  onChangeText,
  placeholder,
  error,
  secureTextEntry,
  rightIcon = null,
  onIconPress,
  inputStyle = {},
}) {
  const inputName = label.toLowerCase().replace(/\s+/g, "_");

  return (
    <View style={{ marginBottom: 16 }}>
      {/* Label shown above the input field */}
      <Text style={styles.label}>{label}</Text>

      {/* Input field with optional right icon (e.g. eye for password) */}
      <View style={[styles.inputWrapper, error ? styles.inputError : null]}>
        <TextInput
          name={inputName}
          id={inputName}
          style={[styles.input, inputStyle]}
          placeholder={placeholder}
          placeholderTextColor="#999"
          value={value}
          onChangeText={onChangeText}
          secureTextEntry={secureTextEntry}
        />
        {rightIcon && (
          <TouchableOpacity onPress={onIconPress}>{rightIcon}</TouchableOpacity>
        )}
      </View>
```

Figure 21 | TextErrorInput Component – Label, Input, Error Message and Optional Icon

### 5.1.4 | IOS-BUTTON COMPONENT

The <*IOSButton*> component is a reusable button designed to match the look and feel of native iOS buttons. It is built using <*TouchableOpacity*> and wraps a custom <*LabelText*> component to display the button title. The button accepts two props: title, which defines the button text, and onPress, which defines the action to perform when the button is pressed.

On the <LoginScreen>, this component is used to trigger the *handleLogin* function. When the button is pressed, the function validates the username, and password inputs and updates the login state if both fields are valid. By creating the button as a standalone component, the design remains consistent and easier to maintain across the different screens.

The code snippet below shows the full implementation of the <IOSButton> component, it demonstrates how the button is constructed using <TouchableOpacity> and styled with a dark background and rounded corners. The component's use of *title* and *onPress* props makes it easier to reuse in different parts of our application.

```jsx
import { TouchableOpacity, StyleSheet } from "react-native";
import LabelText from "./text/label-text";

// A simple reusable button that looks like IOS-style
// It is used for login, signup etc.
export default function IOSButton({ title, onPress }) {
  return (
    <TouchableOpacity style={styles.button} onPress={onPress}>
      <LabelText variant="light">{title}</LabelText>
    </TouchableOpacity>
  );
}

// Styling til knappen
const styles = StyleSheet.create({
  button: {
    justifyContent: "center",
    alignItems: "center",
    width: "100%",
    height: 40,
    backgroundColor: "#1e1c19",
    borderRadius: 8,
  },
});
```

Figure 22 | IOSButton Component – Reusable Login Button

## 5.2 | FEED SCREEN – NINA

The <*FeedScreen*> presents an interactive overview consisting of four information cards. Each card opens a full-screen overlay with extended content. This screen is designed to support learning and exploration while maintaining a consistent visual identity throughout the application.

To manage content visibility, the screen uses state to control which overlay is active. When a card is pressed, a fade-out animation is triggered using the <Animated> API, controlled by the *useEffect* hook in combination with *Animated.timing*. This creates a smooth transition effect as the overlay appears.

The *AboutCard* is an inline component used to display each individual card. It includes an image, a title, and a “Read more” prompt. Each card is <Pressable> component and triggers a specific overlay when pressed. This structure ensures reusability and a clean UX.

The <IOSButton> component is reused on all four overlays to allow users to close the modal. This button is described in more detail in section 5.1.4: *IOSButton Component*.

The code snippet below shows the definition of the *AboutCard* component, which is reused to display each of the four cards on the feed screen.

```
// This is a reusable card component that we use 4 times
const AboutCard = ({ imageSource, title, onPress }) => (
  // Pressable makes the whole card clickable
  <Pressable onPress={onPress} style={styles.card}>
    {/* Image at the top of the card */}
    <Image source={imageSource} style={styles.cardImage} />

    {/* Text content below the image */}
    <View style={styles.cardContent}>
      <Text style={styles.cardTitle}>{title}</Text>
      <Text style={styles.cardReadMore}>Read more</Text>
    </View>
  </Pressable>
);
```

Figure 23 | AboutCard Component - reusable pressable card with image & text

### 5.1.2 | OVERLAY-RECIPE COMPONENT

The <*OverlayRecipe*> component displays a full-screen overlay containing a recipe and visual content. The screen opens with a smooth fade and zoom-in animation to create visual emphasis. At the top of the overlay, a hero image is displayed with an overlaid title. Below that, an animated yellow content box displays a picture of the dish, along with the recipe title, description, ingredients, and instructions. The custom <*IOSButton*> at the bottom allows the user to close the overlay (5.1.4: *IOSButton Component)*

The Animations are managed using React Native's *Animated* API, and the *useEffect* hook ensures that the fade-in animation begins automatically when the overlay appears on the screen.

### 5.1.3 | OVERLAY-FAQ COMPONENT

The <*OverlayFAQ*> component consists of a scrollable FAQ interface that combines predefined questions and answers with user input. Each question is displayed in an expandable block. When the user taps a question, the corresponding answer appears below using a toggle interaction, improving readability and interaction flow.

An additional input field makes the overlay more interactive by allowing users to submit their own questions. Upon submission, a confirmation message appears with a placeholder response.

Animated transitions are triggered when the overlay opens, and a zoom-in and fade-in effect to draw focus to the yellow content box. The interaction within the overlay includes a toggle function for showing or hiding the answers. After submitting a new question, a confirmation message is displayed. A custom <IOSButton> is positioned at the bottom of the screen to allow users to close the overlay.

The code snippet below demonstrates how each FAQ item is rendered using a loop, allowing users to tap a question to expand or collapse the answer. The toggle symbol (+/-) and conditional display of the answer are managed using component state, creating a clear and user-friendly interaction.

```
{/* This is the yellow animated box that holds everything: questions + input */}
<Animated.View style={[animatedStyle, styles.yellowBox]}>
  <View style={styles.contentBox}>
    {/* Title of the FAQ section (centered and styled) */}
    <HeaderText align="center">FAQ</HeaderText>

    {/* Subheading to introduce the questions */}
    <LabelText>Frequently Asked Questions</LabelText>

    {/* Loop through each question and render it */}
    {questions.map((item, index) => (
      <View key={index} style={styles.qaBlock}>
        <TouchableOpacity
          onPress={() => toggleExpand(index)}
          style={styles.questionRow}
        >
          {/* This shows a + or - depending on whether it's open */}
          <Text style={styles.toggleSymbol}>
            {expanded === index ? "-" : "+"}
          </Text>
          <Text style={styles.questionText}>{item.question}</Text>
        </TouchableOpacity>

        {/* Only show the answer if this question is expanded */}
        {expanded === index && <BodyText>{item.answer}</BodyText>}

        {/* Line between questions */}
        <View style={styles.separator} />
      </View>
    ))}
```

Figure 24 | OverlayFAQ – Expandable FAQ Interaction

## 5.1.4 | OVERLAY-COFFEE COMPONENT

The <*OverlayCoffee*> component displays educational content about reusing coffee grounds to grow mushrooms. It opens a full-screen overlay and includes a top image followed by an

animated content box containing structured text. The animation uses the *Animated* API to create a smooth fade-in and scale effect when the overlay appears.

The text content is structured using consistent, reusable components: <HeaderText> for the main title, <LabelText> for subheadings, and <BodyText> for explanatory text. This ensures a clear hierarchy and cohesive visual design across the overlays.

The content structure and animation behavior are consistent with the other overlays, ensuring a seamless user experience. The overlay closes using the shared *<IOSButton>,* described earlier in Section 5.1.4.

The code snippet below shows the main content area of the <OverlayCoffee> component. The text is wrapped inside an *Animated.View,* which fades and scales into view.

```
// Define the animation styles (fade-in + slight scale)
const animatedStyle = {
  opacity: animatedValue,
  transform: [
    {
      scale: animatedValue.interpolate({
        inputRange: [0, 1],
        outputRange: [0.95, 1],
      }),
    },
  ],
};

return (
  <View style={styles.screenBackground}>
    <ScrollView contentContainerStyle={styles.overlayContainer}>
      {/* Top image section (no text overlay anymore) */}
      <View style={styles.fullWidthImageWrapper}>
        <Image
          source={require("../../assets/Coffee1.png")}
          style={styles.fullWidthImage}
        />
      </View>

      {/* Animated content box */}
      <Animated.View style={[styles.contentBox, animatedStyle]}>
        {/* NEW: Main title moved from image to inside content box */}
        <HeaderText align="center">Beyond Coffee</HeaderText>
```

Figure 25 | OverlayCoffee – Animated Content Layout with Scrollable Structure

### 5.1.5 | OVERLAY-SUSTAINABLE COMPONENT

The *<OverlaySustainable>* component highlights the application's commitment to sustainability. It presents four thematic sections: *Health & Well-being, Sustainable Cities & Communities, Responsible Consumption & Production, and Climate Action*. Each section includes a bold subheading and explanatory text.

Like the other overlays, it features a hero image at the top and uses an animated entry for visual empathies. The content is wrapped in the same yellow container, and the layout remains scrollable to accommodate various screen sizes.

## 5.3 | MAP SCREEN – JOSEPH

The MapScreen plays a central part of the project. While being a compulsory aspect of the project, it is also the part of the app where the user first gets introduced to, and gets connected with, other users.

The screen itself is rather innocuous. Central to the screen is a <mapview> where two of the central components get parsed: the custom MapMarker as well as the subsequent MapPopup. Above the displayed map is a custom <filter> component that seeks to filter users (displayed as map markers) based on interest.

### 5.3.1 | MAPMARKER COMPONENT

The main purpose of the MapMarker component is to display a custom visual map marker in replacement of the default marker. This is both for visual congruency within the app and to include interchangeable icons that define one of the three user purposes.

```
function MapMarker({
  coordinate,
  diameter = 50,
  fillColor = styles.circle.backgroundColor,
  strokeColor = styles.circle.borderColor,
  strokeWidth = styles.circle.borderWidth,
  icon = null,
  shadowOptions = styles.shadow,
  onPress,
  user
}) {
```

Figure 26 | MapMarker parameters

These parameters are necessary for the display and functionality of the marker. Coordinating is necessary for longitude and latitude for plotting on the map. Icon lets us place an icon in the

marker and onPress handles a function on press. user data between marker and popup are passed through the user. The rest are for styling.

```
return (
  <Marker
    coordinate={coordinate}
    anchor={{ x: 0.5, y: 0.5 }}
    onPress={() => onPress(user)}
  >
    <View style={[
      styles.circle,
      {
        width: diameter,
        height: diameter,
        borderRadius: diameter / 2,
        backgroundColor: fillColor,
        borderColor: strokeColor,
        borderWidth: strokeWidth,
      },
      combinedShadowStyle
    ]}>
      {icon && (
        <View style={styles.iconContainer}>
          {icon}
        </View>
      )}
    </View>
  </Marker>
```

Figure 27 | Marker render

The <marker> component is a default react native component that allows for the placing of a map pin marker on a map view. The marker here parses the coordinates given and places the map reference in the center of the marker. A circular view replaces the default look of the map marker. The parameters reference values defined in the stylesheet. This looks awkward in the code but has been done to keep styling in the stylesheet.

### 5.3.2 | MAPPOPOP COMPONENT

The purpose of the MapPopup is to present the user with a small interactive menu when pressing one of the markers on the map. This is to add further complexity and options to interactions with users on the map without taking the user off the map screen. This component uses the default react component <Modal>. <Modal> is specifically used to hide and show specific views while including attributes like transparency and animation.

```
export default function MapPopup ({ user, visible, onClose }) {
  if (!user) return null;
```

Figure 28 | MapPopup parameters

The component handles the same user data as the MapMarker component ensuring that the popup shown matches the same user as the marker. Visible controls whether the modal is shown; onClose is a callback function to close the modal.

```
return (
  <Modal
    visible={visible}
    transparent={true}
    animationType="fade"
    onRequestClose={onClose}
  >
    <View style={styles.modalOverlay}>
      <View style={styles.popup}>
        <HeaderText>{user.name}</HeaderText>
        <BodyText>
          {user.name} is interested in {user.mainInterest}! Start a chat to
          find an agreement.
        </BodyText>
        <View style={styles.buttonView}>
          <IOSButton
            onPress={() => {
              navigation.navigate("Messages");
              onClose();
            }}
            title="Start chat"
          />
          <IOSButton onPress={onClose} title="Close popup" />
```

Figure 29 | MapPopup render

The <Modal> component sets transparency of the modal and used animationType “fade” for the modal view to fade in when called.

The first view of the component covers the entire screen and is a transparent dark brown colour. Nested within is another view that makes up the actual visible popup box. This is done to still give visibility and the “feel” of being on the map screen while the interactions are on the popup. The popup calls the username and main interest from the locationData list to be displayed in a short description. One of the buttons closes the popup; the other navigates the user to the message screen to begin chatting with the other user. While I wanted the popup to navigate to

the specific chat pertaining to the user selected on the map, I couldn't figure out how to do so. As such it just navigates to the message screen.

### 5.3.3 | FILTERBUTTON COMPONENT

The filter button handles filtering of our purposes by creating a pressable button for each filter value: in our case the three purposes/interests. This is done by having pressable corresponding to each interest. When pressed the button toggles and adds the interest to a list, that can be used for filtering when rendering other components. When the button is untoggled the item is removed from the list.

```
function handleSelectOption() {
  if (selectedFilterOptions.includes(filterOptionValue)) {
    setSelectedFilterOptions(
      selectedFilterOptions.filter(
        (selectedFilterOption) => selectedFilterOption !== filterOptionValue
      )
    );
  } else {
    setSelectedFilterOptions([...selectedFilterOptions, filterOptionValue]);
```

Figure 30 | Main filter function

This function is called upon button press. When pressed it checks if the filterOptionValue is already included in the list. If it is, then it reproduces the lists but without the selected value. If the list does not include the corresponding filterOptionValue, it adds it to the list.

```
<Pressable
  style={[
    styles.filterButton,
    selectedFilterOptions.includes(filterOptionValue) && styles.selected,
  ]}
  onPress={handleSelectOption}
>
  <MaterialCommunityIcons
    name={icon}
    size={24}
    color={
      selectedFilterOptions.includes(filterOptionValue)
        ? "#fefbd9"
        : "#1e1c19"
```

Figure 31 | FilterButton render

The pressable handles the visual feedback of adding items to the list. Upon press the aforementioned function is called and the item is added to the list. The colour also changes. On re-press the value is removed and the colour changes back. The pressable uses a defined style from styles.filterButton and if the corresponding value is included in the list, it includes styles.selected. The same is done when changing colour of the icon.

### 5.3.4 | FILTER COMPONENT

The filter component simply displays the required amount of filter buttons based on the filter data it receives.

```
return (
  <View style={styles.container}>
    {filterOptions.map((filterOption, index) => (
      <FilterButton
        key={index}
        icon={filterOption.icon}
        filterOptionValue={filterOption.value}
        selectedFilterOptions={selectedFilterOptions}
        setSelectedFilterOptions={setSelectedFilterOptions}
```

Figure 32 | Main filter render

The component creates a view where the filter buttons are shown. The position in the list indicates the position of the button in the view. It also parses the icons included in the list.

```
import filterOptions from "../data/filter-options-data";
```

Figure 33 | Filter option file

In this component the data file is imported.

```
const filterOptions = [
  {
    icon: "mushroom",
    value: "Exchanging mushrooms",
```

Figure 34 | Filter option file example

An item in the list looks as so.

### 5.3.5 | THE MAPSCREEN

The map screen itself includes most of the logic required to get the screen working as intended. The screen displays the components mentioned earlier and connects through the <mapview>.

```
const initialRegion = {
  latitude: 55.679768,
  longitude: 12.591411,
  latitudeDelta: 0.0922,
  longitudeDelta: 0.0421,
};
```

Figure 35 | Initial region const

This constant sets the initial region on the map when rendering the <mapview>. These coordinates are set in Copenhagen with a zoom level that ensures all icons are visible. This was done instead of including user location as the screen is meant to be for browsing and not navigating.

```
const [selectedUser, setSelectedUser] = useState(null);
const [popupVisible, setPopupVisible] = useState(false);
```

Figure 36 | Popup hooks

```
function handleMarkerPress(user) {
  setSelectedUser(user);
  setPopupVisible(true);
}

function closePopup() {
  setPopupVisible(false);
  setSelectedUser(null);
}
```

Figure 37 | Popup functions

These functions handle the visibility of the MapPopup component. Each map marker has an onPress that calls the handleMakrerPress function. This selects the respective user and sets visibility of the popup to true. Both buttons on the popup call the closePopup function which removes the popup and deselects a user.

```
const [selectedFilterOptions, setSelectedFilterOptions] = useState([]);
const [renderKey, setRenderKey] = useState(0);
```

Figure 38 | Filter hooks 1

These hooks are important for the functionality of the filter.

```
//force a render of the map on filter press
useEffect(() => {
  setRenderKey(prev => prev + 1);
}, [selectedFilterOptions]);

//keep old filter upon rerender
const updateFilters = useCallback((newFilters) => {
  setSelectedFilterOptions([...newFilters]);
}, []);
```

Figure 39 | Filter hooks 2

useEffect is a react native hook that forces a change through a dependency. In this case, whenever selectedFilterOptions gets updated, the useEffect sets the RenderKey to +1. The RenderKey is called in the <mapview> and forces a rerender of the <mapview>. This is due to the relationship between the filter and the mapmarkers. To display the filtered markers on the map. Re-rendering the map was the solution I was able to implement.

The constant uses a useCallback hook in react native to keep our filters set through re-renders.

```
const filteredMapData = mapLocationData.filter((user) => {
  if (selectedFilterOptions.length === 0) {
    return true;
  }
  return selectedFilterOptions.includes(user.mainInterest);
});
```

Figure 40 | Filter data constant

This constant lets the MapScreen use data from the user list. The first part of the constant checks the length of the filter list. If the length is 0 (no items in the list) it is treated as all filters being selected. The second part filters the user.mainInterest value which gets parsed when rendering the map markers.

```
{filteredMapData.map((user, index) => (
  <MapMarker
    key={`${user.name}-${renderKey}-${index}`}
    user={user}
    icon={
      <MaterialCommunityIcons
        name={user.icon}
        size={36}
        color="#FEFBD9"
      />
    }
    onPress={handleMarkerPress}
    coordinate={{
      latitude: user.latitude,
      longitude: user.longitude,
```

Figure 41 | MapMarker render in mapview

This section is nested in the <mapview> and handles the rendering of the MapMarker component. The first line ensures that only markers matching the filter get rendered. The icon section pulls the user icon from the map-location-data list, which matches the user's main interest. The onPress calls the handleMarkerPress function, which is responsible for displaying the MapPopup. The coordinates are also pulled from the map-location-data list.

```
import mapLocationData from "../data/map-location-data";
```

Figure 42 | mapLocationData file

The user data is located in this file.

```
{
  name: 'Karla25',
  icon: 'mushroom',
  mainInterest: 'Exchanging mushrooms',
  latitude: 55.667161,
  longitude: 12.6292606,
},
```

Figure 43 | mapLocationData example

An item in the list looks as so.

## 5.4 | MESSAGE SCREEN – PARNUNA

MessageScreen brings together several custom hooks to handle complex user interactions efficiently. The useUserFiltering hook (line 17) manages all the filtering logic; search queries, selected interests, and filter resets, in one centralized place rather than cluttering the main component. Meanwhile, useInterestIcons (line 19) handles the dynamic rendering of interest icons, keeping that logic separate and reusable. The handleUserPress function (lines 21-23) provides clean navigation to the chat screen, passing the selected user's name as a parameter to maintain context across screens.

message-screen.jsx

screens > message-screen.jsx > MessageScreen

```
1  import ScreenLayout from "../components/screen-layout";
2  import useUserFiltering from "../hooks/use-user-filtering";
3  import useInterestIcons from "../hooks/use-interest-icons";
4  import FilterSection from "../components/message/filter-section";
5  import SearchBar from "../components/message/search-bar";
6  import UserList from "../components/message/user-list";
7  import userData from "../data/user-data";
8
9  function MessageScreen({ navigation }) {
10   const {
11     filteredUsers,
12     searchQuery,
13     selectedInterests,
14     setSearchQuery,
15     toggleInterest,
16     resetAllFilters,
17   } = useUserFiltering(userData);
18
19   const { renderInterestIcon } = useInterestIcons();
20
21   const handleUserPress = (userName) => {
22     navigation.navigate("Chat", { userName });
23   };
```

Figure 44 | MessageScreen Imports and Hook Configuration

### 5.4.1 | COMPONENTS FOR MESSAGESCREEN

The FilterButton component is to eliminate repetition, it lives in one place (lines 3-21 in filter-button.jsx), making it straightforward to add new filter types or modify button behavior.

```
3   export default function FilterButton({
4     interestKey,
5     isActive,
6     onToggle,
7     renderIcon
8   }) {
9     return (
10      <TouchableOpacity
11        style={[
12          styles.customFilterButton,
13          isActive && styles.customFilterButtonActive
14        ]}
15        onPress={() => onToggle(interestKey)}
16      >
17        <View style={styles.filterButtonIconContainer}>
18          {renderIcon(interestKey, 20, isActive)}
19        </View>
20      </TouchableOpacity>
21    );
```

Figure 45 | Interactive Toggle Filter Button

FilterSection component takes responsibility for the entire filtering interface at the top of the screen. It orchestrates the three filter buttons, displays the current selection count, and handles the reset functionality. The component uses clean prop interfaces (lines 5-10 in filter-section.jsx) to communicate with the parent, keeping the logic predictable and the component boundaries clear.

```
5   export default function FilterSection({
6     selectedInterests,
7     onToggleInterest,
8     onResetFilters,
9     renderInterestIcon
10  }) {
```

Figure 46 | Single responsibility principle and clean prop interfaces

UserItem component handles individual user rendering. Each user profile now knows how to present itself with the right image, name, interest badges, and timestamp. The navigation logic lives here too (line 10 in user-item.jsx), appearing as a simple press handler that communicates back to the parent screen.

SearchBar component manages the text input functionality as a focused, single purpose component. Rather than embedding search logic directly in the main screen, this component (lines 4-15 in search-bar.jsx) handles the input presentation whilst the parent manages the actual filtering state through clean prop passing.

```
4   export default function SearchBar({ searchQuery, onSearchChange }) {
5     return (
6       <View style={styles.searchContainer}>
7         <SearchInput
8           value={searchQuery}
9           onChangeText={onSearchChange}
10          placeholder="Search..."
11          placeholderTextColor="#888"
12          containerStyle={styles.searchInputContainer}
13        />
14      </View>
15    );
```

Figure 47 | Real-time Search Input Component

I started out using a FlatList in the UserList component, but it interfered with our ScrollView, which gave an error VirtualizedList. I then replaced it with map rendering (lines 9-19 in user-list.jsx), eliminating the scroll conflicts that were plaguing the interface. This component now works with ScreenLayout's ScrollView.

```
9     return (
10      <View style={styles.listContainer}>
11        {users.map((user) => (
12          <UserItem
13            key={user.id}
14            user={user}
15            onPress={onUserPress}
16            renderInterestIcon={renderInterestIcon}
17          />
18        ))}
19      </View>
```

Figure 48 | User List Rendering Logic

## 5.4.2 | HOOKS FOR MESSAGESCREEN

The main MessageScreen manages the overall state through custom hooks and orchestrates communication between components. The entire component now reads like a simple composition of focused parts.

My custom hooks transform state management. The useUserFiltering hook (lines 47-59 in use-user-filtering.js) encapsulates all the complex filtering logic that was previously cluttering the main component. When I call this hook with my users array, I get back a clean API with filteredUsers, searchQuery, selectedInterests, and all the action functions I need. No more managing multiple useState calls or complex filtering logic in the component itself (react.dev, 2025).

```
47    return {
48      // State værdier / State values
49      searchQuery,
50      selectedInterests,
51
52      // Filtering funktioner / Filtering functions
53      filteredUsers: getSearchFilteredUsers(),
54
55      // Action funktioner / Action functions
56      toggleInterest,
57      resetAllFilters,
58      setSearchQuery: setSearchQueryValue,
59    };
60  }
```

Figure 49 | Custom hooks encapsulate complex logic

The useInterestIcons hook centralises all icon rendering logic. Previously, I had the same icon mapping code repeated wherever I needed to display coffee cups, mushrooms, or helping hands. Now this logic lives in one place (lines 13-38 in use-interest-icons.js), handling both filter button contexts and user list displays through a single, consistent interface.

```
// Renderer for ikoner / Renderer for icons
function renderInterestIcon(interestKey, size = 16, isActive = false) {
  const icon = interestIcons[interestKey];

  // Tjek om ikonet eksisterer / Check if icon exists
  if (!icon) {
    console.warn(`Icon not found for interest: ${interestKey}`);
    return null;
  }

  const IconComponent = icon.component;

  // For filter-knapper (øverst) / For filter buttons (top)
  if (isActive !== false && typeof isActive === 'boolean') {
    return (
      <IconComponent
        name={icon.name}
        size={size}
        color={isActive ? '#FEFBD9' : '#000'}
      />
    );
  }

  // For bruger-liste ikoner / For user list icons
  return <IconComponent name={icon.name} size={size} color="#222" />;
}
```

Figure 50 | Dynamic Interest Icon Component

### 5.4.3 | BEST PRACTICES CODING MESSAGESCREEN

Throughout this coding process, I have stuck to several key principles that make the code readable and maintainable. Function names like toggleInterest and resetAllFilters tell you exactly what they do briefly. Each component handles one clear job - FilterSection doesn't mess with user display, and UserItem stays focused on showing user info rather than dealing with search logic.

I followed the DRY principle religiously. Instead of scattering user data everywhere or repeating interest configurations, I created reusable constants in dedicated files. When I hit errors during development, Claude AI helped me debug issues and refine my approach. User data gets imported from constants/user-data.js along with helper functions like getUserById, while interest-config.js centralizes all the icon mappings and color schemes. My constants files follow consistent patterns - app-styles.js provides design tokens for colors, spacing, and shadows, keeping everything visually cohesive.

This structure makes the app more responsive because each component can optimize its own rendering. When someone types in the search bar, only SearchBar updates. Same goes for filter buttons, only FilterSection changes immediately, then the filtering hook processes everything and updates the user list. Testing became much more straightforward. I used Claude AI as a sparring partner to identify edge cases and ensure each component worked independently, making it easier to catch problems and verify that UserItem displays correctly regardless of whether the search filtering is working properly.

There are areas where we could push this further. Adding loading states for user data fetching would improve the user experience. The filter buttons could show subtle animations during state transitions, and the search could implement debouncing to improve performance during rapid typing.

## 5.5 | CHAT SCREEN – PARNUNA

My ChatScreen is broken down into focused, reusable pieces that each have a clear purpose. I have created distinct components, each living in the components/chat/ directory and handling one specific aspect of the chat interface. I have organized my helper functions in dedicated utils files like chatHelper.js, which handles message initialization and time formatting logic. ProfileContext provides centralized user management across the chat feature, letting components access and update user data without drilling through multiple levels of the component tree.

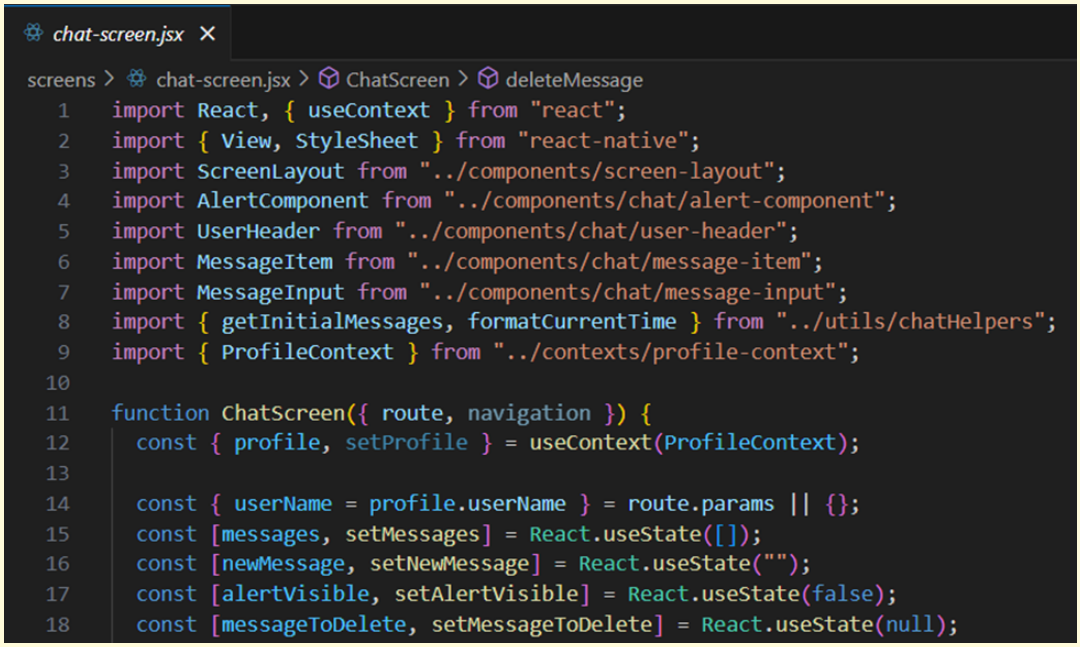


Figure 51 | ChatScreen Imports and State Management

### 5.5.1 | COMPONENTS FOR CHATSCREEN

The InterestIcon component centralizes all icon rendering logic. Previously, I had the same switch statement repeated across different parts of the code whenever I needed to display coffee cups, mushrooms, or helping hands icons. Now this logic lives in one place (lines 11-23 in interest-icon.jsx), making it straightforward to add new interest types without hunting through multiple files.

```
11  switch(icon.component) {
12    case "FontAwesome":
13      IconComponent = FontAwesome;
14      break;
15    case "MaterialCommunityIcons":
16      IconComponent = MaterialCommunityIcons;
17      break;
18    case "FontAwesome5":
19      IconComponent = FontAwesome5;
20      break;
21    default:
22      IconComponent = FontAwesome;
23  }
```

Figure 52 | Icon Library Switch Logic

UserHeader components take responsibility for everything at the top of the screen. It pulls together the user's name, their Copenhagen address, and their purpose statement, along with displaying their relevant interest icons. The component uses my centralized helper functions (lines 20-21, 29, and 36 in user-header.jsx) to fetch the right interests for each user, keeping the logic clean and predictable.

```
20          {getUserInterests(userName).map((interest, index) => (
21            <View key={index} style={styles.headerInterestBadge}>
29                text={getUserAddress(userName)}
36                text={getUserPurpose(userName)}
```

Figure 53 | UserHeader: Centralized User Display Logic

MessageItem component handles individual message rendering, which turned out to an organizational win. Each message now knows how to present itself, whether it's been sent by you or received from someone else (lines 8-16 in message-item.jsx). The delete functionality

lives here too (lines 58-69 in message-item.jsx), appearing only on messages you've sent. This separation means the message display logic doesn't interfere with other parts of the screen.

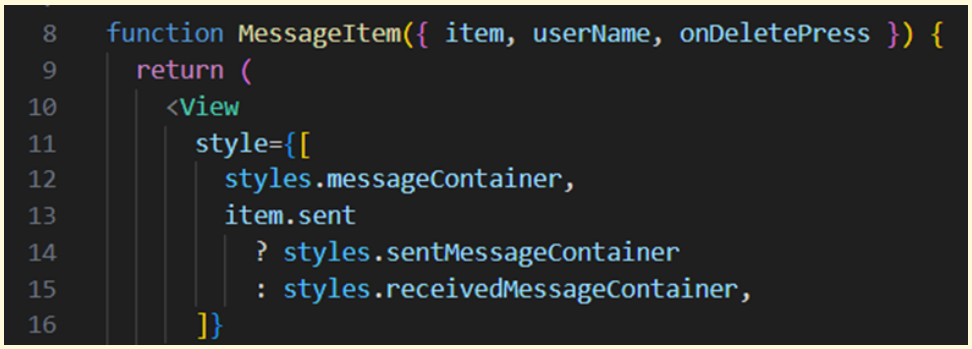


Figure 54 | Component knows how to present itself

MessageInput component manages the text input field and send button. As a controlled component, it focuses purely on presentation and user interaction whilst the parent component handles the actual state. The disabled button logic (lines 23-30 in message-input.jsx) demonstrates how clean prop handling can keep component boundaries clear.

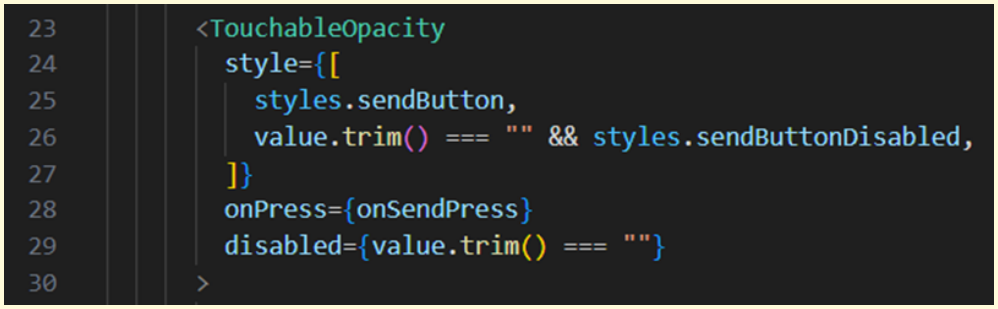


Figure 55 | Conditional Send Button Logic

AlertComponent costume component handles all our modal dialogs in one place. Instead of writing modal setup code every time we need a confirmation dialog, this component takes care of the presentation while the parent handles what happens when someone confirms or cancels. The default prop values (lines 8-9 in alert-component.jsx) mean we can drop it anywhere and customize the text without rebuilding the entire modal structure each time.

```
3   function AlertComponent({
4     visible,
5     title,
6     message,
7     onConfirm,
8     onCancel,
9     confirmText = 'Bekræft',
10    cancelText = 'Annuller'
11  }) {
12    return (
13      <Modal
14        transparent={true}
15        visible={visible}
16        animationType="fade"
17        onRequestClose={onCancel}
18      >
```

Figure 56 | Prop interface and Modal setup

The main ChatScreen is a coordinator; it manages the overall state and orchestrates communication between components. The renderMessage function (lines 66-75) now simply passes props to the appropriate component rather than handling all the display logic itself.

```
66          <View style={styles.messageList}>
67            {messages.map((message) => (
68              <MessageItem
69                key={message.id}
70                item={message}
71                userName={userName}
72                onDeletePress={showDeleteAlert}
73              />
74            ))}
75          </View>
```

Figure 57 | Message List Rendering Logic

## 5.5.2 | BEST PRACTICES CODING FOR CHATSCREEN

In this screen I have used the simple code principle, so function names like getUserInterests and formatCurrentTime in our chatHelpers.js file immediately tell you what they're supposed to do.

I built reusable utilities in my helpers file following DRY principles. When getUserInterests ("Claus47") gets called, it consistently returns the same ["coffee-cup"] array whether UserHeader or MessageItem needs it, cutting out duplicate mapping logic across components.

Rather than creating deeply nested components that become hard to navigate, I kept the structure flat and manageable. My naming stays consistent, data-fetching functions start with "get", event handlers begin with "on", and state setters use descriptive names that make their purpose obvious. I focused my comments on explaining the why behind business logic instead of stating what's already clear from reading the code. When I ran into tricky rendering issues, Claude AI helped me identify performance bottlenecks and optimize the component lifecycle.

The flat architecture keeps things snappy since each component handles its own rendering independently. When someone types in the message input, only that specific component updates instead of forcing the entire chat screen to re-render. Testing becomes much cleaner too - I can verify UserHeader shows the right user info separate from testing whether messages send properly, which makes hunting down bugs and edge cases way more manageable.

## 5.6 | HOME SCREEN – VICTOR

The home screen aims to bring together all the features that have to do with customizing your profile, and for this reason contains many different types of features to customize your profile according to your own wishes, which I have divided into four different sections.

### 5.6.1 | PROFILE IMAGE COMPONENT

The first section is intended to display some user identification in the form of the user's profile image and username using the <ProfileImage> and the <HeaderText> components.

```
37          <View style={styles.sectionTypeOne}>
38            <ProfileImage profileImage={profile.profileImage} />
39
40            <HeaderText align="center">{profile.username}</HeaderText>
41          </View>
```

Figure 58 | First section in the home screen

I made the <ProfileImage> component myself as a reusable component with a profileImage prop used to display the profile image using an <Image> component from the React Native framework.

```
//Profile image component
export default function ProfileImage({ profileImage }) {
  const navigation = useNavigation();

  return (
    <Pressable
      style={styles.profileImage}
      onPress={() =>
        //Navigates to the camera screen when pressed
        navigation.navigate("Camera")
      }
    >
      <Image
        source={
          //Checks if a profile image exists
          profileImage
            ? { uri: profileImage }
            : require("../../assets/icon.png")
        }
        style={styles.image}
      />

      <View style={styles.cameraIcon}>
        <MaterialCommunityIcons name="camera" size={32} color="#1e1c19" />
      </View>
    </Pressable>
  );
}
```

Figure 59 | Profile image component

Additionally, I have also made it possible to navigate to the camera screen to take a photo and replace the current profile image when the <ProfileImage> component is pressed. This is done by using the useNavigation hook from the React Navigation library to access the navigation object, which is then used to set up the navigation to the camera screen in the onPress prop of the <Pressable> components, which is the parent component of the <ProfileImage> component.

## 5.6.2 | PURPOSELIST AND PURPOSE ITEM COMPONENTS

The second section of the home screen aims to get the user to indicate why they are using the mobile application so that the user can be displayed on the map screen and their purpose can be seen by other users using a <Filter> component.

```
<View style={styles.sectionTypeTwo}>
  <InfoBox infoText="The purpose(s) you select will be displayed to others as posts on the map.">
    <HeaderText>Purpose</HeaderText>
  </InfoBox>

  <BodyText>Select one or multiple purposes:</BodyText>

  <PurposeList filterOptions={filterOptions} />

  <Filter
    selectedFilterOptions={selectedFilterOptions}
    setSelectedFilterOptions={setSelectedFilterOptions}
  />
</View>
```

Figure 60 | Second section in the home screen

To help explain how the filter works, I have created both a <PurposeList> and a <PurposeItem> components that help explain the meaning of the icons in the <FilterButton> components inside of the <Filter> component.

```
//Purpose list component
export default function PurposeList({ filterOptions }) {
  return (
    <View style={styles.purposeList}>
      {
        //Renders a purpose item component for each of the filter options
        filterOptions.map((filterOption, index) => (
          <PurposeItem
            key={index}
            label={filterOption.value}
            icon={filterOption.icon}
          />
        ))
      }
    </View>
  );
}
```

Figure 61 | Purpose list component

The <PurposeList> uses the filterOptions prop to render a <PurposeItem> component for each of the individual filter options to explain the meaning using the Array map method.

The <PurposeItem> is a reusable component with a label and an icon prop. The icon prop is used in the name prop of a <MaterialCommunityIcons> component from the Expo Vector icon library to display the icon for the associated filter button, and the label prop is used in a <LabelText> component to act as explanatory text.

```
5   //Purpose item component
6   export default function PurposeItem({ label, icon }) {
7     return (
8       <View style={styles.purposeItem}>
9         <MaterialCommunityIcons name={icon} size={32} color="#1e1c19" />
10
11        <LabelText>{label}</LabelText>
12      </View>
13    );
14  }
```

Figure 62 | Purpose item component

## 5.6.3 | LOCATION ADDRESS COMPONENT

The third section of the home screen tells the user what their current location address is and where they will be displayed on the map screen for the other users if they have chosen to indicate why they are using the mobile application in the second section using a <LocationAddress> component I created.

```
58            <View style={styles.sectionTypeTwo}>
59              <InfoBox infoText="Others will be able to see your posts at your location address if you don't hide it.">
60                <HeaderText>Location address</HeaderText>
61              </InfoBox>
62
63              <LocationAddress />
64            </View>
```

Figure 63 | Third section in the home screen

I have set up a handleGetLocation function inside the <LocationAddress> component to get the users' current location.

```
16  async function handleGetLocation() {
17    //Gets the status of location permissions
18    let { status } = await Location.requestForegroundPermissionsAsync();
19
20    //Checks if location permissions is granted
21    if (status !== "granted") {
22      //Assigns an error message and returns it if location permissions have not been granted yet
23      setErrorMsg(
24        "Permission to access location was denied, others will not be able to see your posts on the map"
25      );
26
27      return;
28    }
29
30    //Gets the location data
31    let location = await Location.getCurrentPositionAsync({});
32
33    //Saves the location
34    setLocation(location);
35  }
```

Figure 64 | The handleGetLocation function in the location address component

The first thing that happens in the function is to retrieve the status of the location permissions using the requestForegroundPermissionsAsync function from the expo location library to check if the permissions are granted using a conditional if statement. If the location permissions are not yet granted, an error message will be configured and displayed using the useState hook from the React framework. If the location permissions are granted, the handleGetLocation function will get the user's current location data using the getCurrentPositionAsync function from the expo location library and store it using the useState hook from the React framework.

To get the current location of the user, when the <LocationAddress> component is rendered, I have placed the handleGetLocation function in a useEffect hook from the React framework with an empty dependency array inside.

```
37  useEffect(() => {
38    //Gets the location data when the component is renders
39    handleGetLocation();
40  }, []);
```

Figure 65 | The first useEffect hook used in the location address component

I have also created a handleGetAddres function inside the <LocationAddress> component to get the matching current address for the user's current location data. The first thing that happens in the function is that it retrieves the user's current address using the user's current location data and the reverseGeocodeAsync function from the expo location library and configures it using the useState hook from the React framework.

```
42  async function handleGetAddress() {
43    //Gets the address matching the location data
44    let address = await Location.reverseGeocodeAsync({
45      latitude: location.coords.latitude,
46      longitude: location.coords.longitude,
47    });
48
49    //Saves the address matching the location data
50    setLocationAddress(
51      `${address[0].street} ${address[0].streetNumber}, ${address[0].postalCode} ${address[0].city}, ${address[0].country}`
52    );
53  }
```

Figure 66 | The handleGetAddress function in the location address component

To get the user's current address when they are moving location, I have placed the handleGetAddress function in a useEffect hook from the React framework with the current location data state value that was stored using the useState hook from the React framework inside its dependency array.

```
55    useEffect(() => {
56      //Gets the addrees when the location changes
57      handleGetAddress();
58    }, [location]);
```

Figure 67 | The second useEffect hook used in the location address component

## 5.6.4 | INFO BOX COMPONENT

In both the second and third sections of the home screen, additional information is needed to explain their complexity, which is why I have created the <InfoBox> component.

```
7   //Info box component
8   export default function InfoBox({ infoText, children }) {
9     const [showInfo, setShowInfo] = useState(false);
10
11    return (
12      <View style={styles.infoBox}>
13        <View style={styles.infoProvider}>
14          {children}
15
16          <Pressable onPress={() => setShowInfo(!showInfo)}>
17            <MaterialCommunityIcons
18              name={
19                //Checks if the info is shown or not
20                showInfo ? "information" : "information-outline"
21              }
22              size={32}
23              color="#1e1c19"
24            />
25          </Pressable>
26        </View>
27
28        {
29          //Checks if the info is shown or not
30          showInfo && (
31            <View style={styles.infoText}>
32              <LabelText>Info:</LabelText>
33
34              <BodyText>{infoText}</BodyText>
35            </View>
36          )
37        }
38      </View>
39    );
40  }
```

Figure 68 | Info box component

The <InfoBox> component is a reusable component with an infoText and a children prop. The infoText prop is used inside a <BodyText> component to write some additional information, and the children prop is used to wrap the <InfoBox> component around some kind of text

component, such as the <HeaderText> components in the second and third sections on the home screen.

To make the <InfoBox> component act as a tooltip that displays additional information when the information icon is pressed, the state of the additional information is configured using the useState hook from the React framework inside the onPress prop of the <Pressable> component, which is wrapped around the <MaterialCommunityIcons> icon component from the Expo Vector icon library to  make it act like a switch when pressed, hiding and showing the additional information.

### 5.6.5 | SLIDER, SLIDER ITEM & PAGINATION COMPONENTS

The fourth section of the home screen is used to navigate users to the map screen via a <Slider> component I created so they can explore the different map markers on the map.

```
66          <View style={styles.sectionTypeThree}>
67            <Slider data={sliderdata} />
68          </View>
```

Figure 69 | Fourth section in the home screen

The <Slider> component takes a data prop that is passed to a <FlatList> component from the React Native framework, and a <Pagination> component that I created myself. The data prop from the <Slider> component is used in the <FlatList> component to render the <SliderItem> components in the <Slider> component, and the data prop from the <Slider> component is used in the <Pagination> component to render a number of dots that matches the number of <SliderItem> components in the <Slider> component.

```jsx
//Pagination component
export default function Pagination({ data, scrollX }) {
  const { width } = useWindowDimensions();

  return (
    <View style={styles.container}>
      {data.map((_, index) => {
        //Specifies the input range for the pagination animation, which corresponds to the previous dot, the curren dot and the next dot
        const inputRange = [
          (index - 1) * width,
          index * width,
          (index + 1) * width,
        ];

        //Animates the width of the pagination dots
        const dotWidth = scrollX.interpolate({
          inputRange,
          outputRange: [10, 20, 10],
          extrapolate: "clamp",
        });

        //Animates the opacity of the pagination dots
        const dotOpacity = scrollX.interpolate({
          inputRange,
          outputRange: [0.5, 1, 0.5],
          extrapolate: "clamp",
        });

        return (
          <Animated.View
            key={index}
            style={[styles.dot, { width: dotWidth, opacity: dotOpacity }]}
          />
        );
      })}
    </View>
  );
}
```

Figure 70 | Pagination component

Both the <Slider> and the <Pagination> components are animated using the Animated component from the React Native library.

```jsx
  return (
    <View style={styles.container}>
      <View style={{ flex: 3 }}>
        <FlatList
          data={data}
          renderItem={({ item }) => <SliderItem item={item} />}
          horizontal
          showsHorizontalScrollIndicator={false}
          pagingEnabled
          onScroll={
            //Animates the slider when scrolled
            Animated.event(
              [{ nativeEvent: { contentOffset: { x: scrollX } } }],
              { useNativeDriver: false }
            )
          }
          onViewableItemsChanged={viewableItemsChanged}
          ref={slidesRef}
        />
      </View>

      <Pagination data={data} scrollX={scrollX} />
    </View>
  );
}
```

Figure 71 | The building blocks of the slider component

I created a viewableItemsChanged constant that I used as the value of the onViewableItemsChanged prop for the <FlatList> component inside the <Slider> component to track the state of the index of the currently visible <SliderItem> component using the useRef and the useState hooks from the React framework.

```
14    //Gets the index of the currently visible slider item
15    const viewableItemsChanged = useRef(({ viewableItems }) => {
16      setCurrentIndex(viewableItems[0].index);
17    }).current;
```

Figure 72 | The viewableItemsChanged constant

I have then created a scrollTo function that allows you to change the currently visible <SliderItem> component inside the <Slider> component.

```
19    //A function that allows you to change the currently visible slider item inside the slider
20    function scrollTo() {
21      if (currentIndex < data.length - 1) {
22        slidesRef.current.scrollToIndex({ index: currentIndex + 1 });
23      } else {
24        slidesRef.current.scrollToIndex({ index: 0 });
25      }
26    }
```

Figure 73 | The scrollTo function used in the slider component

To make both the <Slider> and <Pagination> components come to live, I have specified a time interval for when the <Slider> component should call the scrollTo function changing the currently visible <SliderItem> component in a useEffect hook from the React framework with the current index state value stored using the useState hook from the React framework inside its dependency array.

```
28    //Makes the slider move between the slider items in the slider at an interval of 3 seconds
29    useEffect(() => {
30      const interval = setInterval(() => {
31        scrollTo();
32      }, 3000);
33
34      return () => clearInterval(interval);
35    }, [currentIndex]);
```

Figure 74 | The useEffect in the slider component

The <SliderItem> is a reusable component with an item prop. The item prop is an object with multiple values inside. The icon value of the item prop is used in the name prop of a

<MaterialCommunityIcons> component from the Expo Vector icon library to display a unique icon for each of the <SliderItem> components. To display a unique title for each of the <SliderItem> components, the title value of the item prop is used inside a <HeaderText> component. The description value of the item prop is used inside a <BodyText> component to provide a unique description for each of the <SliderItem> components.

I have made it possible to navigate to the map screen when the <IOSButton> component in the <SliderItem> component is pressed. This is done by using the useNavigation hook from the React Navigation library to access the navigation object, which is then used to set up the navigation to the map screen in the onPress prop for the <IOSButtom> component.

```
10  //Slider item component
11  export default function SliderItem({ item }) {
12    const navigation = useNavigation();
13
14    return (
15      <View style={styles.container}>
16        <View style={styles.content}>
17          <View style={styles.info}>
18            <MaterialCommunityIcons name={item.icon} size={64} color="#1e1c19" />
19
20            <HeaderText align="center">{item.title}</HeaderText>
21
22            <BodyText align="center">{item.description}</BodyText>
23          </View>
24
25          <IOSButton
26            title="Explore now"
27            onPress={() =>
28              //Navigates to the map screen when pressed
29              navigation.navigate("Map")
30            }
31          />
32        </View>
33      </View>
34    );
35  }
```

Figure 75 | Slider item component

## 5.7 | CAMERA SCREEN – VICTOR

The camera screen is used to access the camera on the user's device and take a photo to use as the user's profile image in the <ProfileImage> component in the first section of the home screen.

```
46    return (
47      <SafeAreaView style={styles.screen}>
48        {
49          //Checks if a photo is taken or not
50          !photo ? (
51            <Camera setPhoto={setPhoto} />
52          ) : (
53            <Preview photo={photo} setPhoto={setPhoto} />
54          )
55        }
56      </SafeAreaView>
57    );
```

Figure 76 | The building blocks of the camera screen

Inside of the camera screen, a ternary operator is used to check whether a photo has been taken by checking the value of the photo state, which is configured using the useState hook from the React framework. If a photo is taken, a <Preview> component is rendered taking the value of the photo state and the matching set state action as props, allowing the user to check the taken photo, save it to their device in the media library, delete it to replace it with a new photo, or use it as their profile image. If no photo has been taken, a <Camera> component is rendered taking the set state action of the photo state as a prop so that the user can take a photo to use as their profile image.

### 5.7.1 | PERMISSION COMPONENT

Our mobile application needs the permission to use the camera on the user's device, which are configured using the useCameraPermissions hook from the expo camera component, and the permission to use the media library on the user's device, which are configured using the usePermissions hook from the expo media library library.

To create a sort of loading screen while waiting to check whether both of permissions are granted, a conditional if statement is configured to check both permissions using a return statement inside of the conditional if statement to render a <View> component from the React Native Framework if one or both permissions are in the process of loading.

```
17    if (!cameraPermission || !mediaPermission) {
18      //Permissions are still loading.
19      return <View />;
20    }
```

Figure 77 | The conditional if statement that controls the loading screen depending on the state of the permissions on the camera screen

Once both permissions are loaded, two new conditional if statements are configured to check whether each of the two permissions are granted. Return statements is used inside of both conditional if statements to render permission components <Permission> if the permissions are not yet granted, allowing the user to grant the permissions until the user have granted the permissions.

```
22    if (!cameraPermission.granted) {
23      //Camera permissions are not granted yet.
24      return (
25        <Permission
26          icon="camera"
27          text="We need your permission to use the camera"
28          buttonLabel="Grant camera permission"
29          onPress={requestCameraPermission}
30        />
31      );
32    }
```

Figure 78 | The permission check for the camera permissions on the camera screen

```
34    if (!mediaPermission.granted) {
35      //Media library permissions are not granted yet.
36      return (
37        <Permission
38          icon="image-multiple"
39          text="We need your permission to save photos on your device"
40          buttonLabel="Grant storage permission"
41          onPress={requestMediaPermission}
42        />
43      );
44    }
```

Figure 79 | The permission check for the media library permissions on the camera screen

The <Permission> component is a reusable component used to grant the camera permission and the media library permission in the mobile application.

```
6   //Permission component
7   export default function Permission({ icon, text, buttonLabel, onPress }) {
8     return (
9       <SafeAreaView style={styles.permission}>
10        <View style={styles.permissionContainer}>
11          <MaterialCommunityIcons name={icon} size={64} color="#1e1c19" />
12
13          <LabelText align="center">{text}</LabelText>
14
15          <IOSButton title={buttonLabel} onPress={onPress} />
16        </View>
17      </SafeAreaView>
18    );
19  }
```

Figure 80 | The permission component

To make the <Permission> component reuseable it takes the four following props: icon, text, buttonLabel, onPress.

The icon prop is used as a value for the name prop of the <MaterialCommunityIcons> component from the Expo Vector Icon library to render the correct icon for the permissions.

To write user-friendly text in the <Permission> component, the text prop is used in a <LabelText> component, which is inside the <Permission> component.

Both the buttonLabel and onPress props are used for the <IOSButton> component in the <Permission> component, which is the button that grants permission when pressed. The buttonLabel is used as the value of the title prop for the <IOSButton> component to write the label text for the button, and the onPress prop is used as the value of the onPress prop for the <IOSButton> component to grant permission functionality to the <IOSButton> component when pressed.

### 5.7.2 | CAMERA COMPONENT

The <Camera> component is used to make it possible for the user to take a photo to use as their profile image in the mobile application.

```
27    return (
28      <View style={styles.camera}>
29        <View style={styles.cameraViewContainer}>
30          <CameraView style={styles.cameraView} facing={facing} ref={cameraRef} />
31
32          <IconButton
33            icon="camera-flip"
34            buttonLabel="Flip"
35            style={styles.flipButton}
36            onPress={handleToggleCameraFacing}
37          />
38        </View>
39
40        <View style={styles.captureButtonContainer}>
41          <TouchableOpacity
42            style={styles.captureButtonOuter}
43            onPress={handleTakePhoto}
44          >
45            <View style={styles.captureButtonInner} />
46          </TouchableOpacity>
47        </View>
48      </View>
49    );
50  }
```

Figure 81 | The building blocks of the camera component

The <Camera> component consists of a <View> component from the React Native framework, which is wrapped around a camera view container in the form of a <View> component and a capture button container in the form of another <View> component.

Inside the camera view container there is <CameraView> component from the expo camera component and an icon button component <IconButton>. The capture button container is wrapped around a capture button, which is made up of a <TouchableOpacity> component from the React Native framework with another <View> component inside. All the <View> components are for styling purposes only.

To take a photo using the <Camera> component, I created a handleTakePhoto function and assigned it as the value of the onPress prop of the <TouchableOpacity> component, which means that a photo is taken when the capture button in the <Camera> component is pressed.

```
11    async function handleTakePhoto() {
12      //Checks if the camera is ready to take a photo
13      if (cameraRef.current) {
14        //Captures a photo and returns an object containing the photo URI
15        const photo = await cameraRef.current.takePictureAsync();
16
17        //Saves the URI of the captured photo
18        setPhoto(photo.uri);
19      }
20    }
```

Figure 82 | The handleTakePicture function in the camera component

The handleTakePicture function has a conditional if statements inside of it checking if the camera is ready to take a photo by checking the current value of the cameraRef assigned to the <CameraView> component and is configured using the useRef hook from the React framework. If the camera is ready, it will take a photo and store the URI of the captured photo using the setPhoto prop in the <Camera> component.

I have also implemented a flip functionality to the camera, which makes it possible to toggle between the back camera and front camera of the user's device by assigning the handleToggleCameraFacing function as the value of the onPress prop of the <IconButton> component, meaning that the camera flips when the <IconButton> component is pressed.

```
22    function handleToggleCameraFacing() {
23      //Toggles the active camera between the back camera and the front camera
24      setFacing((current) => (current === "back" ? "front" : "back"));
25    }
```

Figure 83 | The handleToggleCameraFacing function in the camera component

In the handleToggleCameraFacing function a set state action is set using the useState hook from the React framework with an arrow function inside with the current camera facing as a parameter. To change the state of the camera facing so the camera flips when the <IconButton> component is pressed, a ternary operator is placed inside of the arrow function that checks the state of the current camera facing and changes the current camera facing to the opposite.

### 5.7.3 | PREVIEW COMPONENT

I have created the <Preview> component for multiple purposes in the mobile application. It is used to check the photo taken with the camera component, save the photo taken on the user's device in the media library, delete the taken photo to replace it with a new photo or use the taken photo as the user's profile image.

Most of those functionalities are made possible using a combination of the useState and useContext hooks from the React framework, and the functionality behind them are pretty self-explanatory. However, to save the taken photo to the media library on the user's device, I created a handleSavePhoto function that is used as the value of the onPress prop of the <Pressable> component the download button in the <Preview> component is made with.

```
18  async function handleSavePhoto() {
19    try {
20      //Saves the photo to the devices's media library
21      await MediaLibrary.saveToLibraryAsync(photo);
22
23      //Updates the value of the saved state
24      setSaved(true);
25    } catch (error) {
26      //Displays an alert, when an error occurs
27      Alert.alert("Failed to save photo.");
28    }
29  }
```

Figure 84 | The handleSavePhoto function in the preview component

In the handleSavePhoto function, a try-and-catch statement is used to save the taken photo to the media library on the user's device using the saveToLibraryAsync function from the Expo media library inside of the try block of the try-and-catch statement. The catch block of the try-and-catch statemen is set to catch potential errors and return an alert using the <Alert> component from the React Native framework with an error message inside.

## 5.7.4 | ICON BUTTON COMPONENT

The <IconButton> component is a reusable button component I have created to be used for multiple purposes in the components inside the camera screen of the mobile application.

```
5   //Icon button component
6   export default function IconButton({ icon, buttonLabel, style, onPress }) {
7     return (
8       <Pressable style={[styles.cameraIconButton, style]} onPress={onPress}>
9         <MaterialCommunityIcons name={icon} size={32} color="#fefbd9" />
10
11        <LabelText variant="light">{buttonLabel}</LabelText>
12      </Pressable>
13    );
14  }
```

Figure 85 | The icon button component

To make the <IconButton> component reuseable it takes the four following props: icon, buttonLabel, style, onPress.

The icon prop is used as a value for the name prop of the <MaterialCommunityIcons> component from the Expo Vector Icon library to render the correct icon for the <IconButton> component.

To write the label text for the <IconButton> component the buttonLabel prop is used inside the <LabelText> component of the <IconButon> component.

Both the style and onPress props are used for the <Pressable> component, which is kind of the <IconButton> component itself. The style prop is used to add more styles in the style prop of the <Pressable> component, and the onPress prop is used as the value of the onPress prop of the <Pressable> component to give some functionality to the <IconButton> component when pressed.

## 5.8 | NAVIGATION – VICTOR

To enable navigation between our screens in our mobile application and ensure a good user flow, I have set up the mobile application's authentication functionality and set up and connected two types of navigation systems (Native Stack and Bottom Tabs) using the React Navigation library.

### 5.8.1 | AUTHENTICATION

I have configured the core authentication functionality in our mobile application in the App.jsx file.

```
7   export default function App() {
8     const [isLoggedIn, setIsLoggedIn] = useState(false);
9
10    return (
11      <ProfileContextProvider>
12        {
13          //Checks if the user is logged in or not
14          isLoggedIn ? (
15            <NavigationContainer>
16              <RootStack />
17            </NavigationContainer>
18          ) : (
19            <LoginScreen setIsLoggedIn={setIsLoggedIn} />
20          )
21        }
22      </ProfileContextProvider>
23    );
24  }
```

Figure 86 | The App.jsx file

The authentication functionality works using a ternary operator that checks the boolean value of the isLoggedIn state, which is configured using the useState hook from the React framework and conditionally renders the <LoginScreen> component or the <NavigationContainer>

component from the React Navigation library. The <NavigationContainer> component is wrapped around the <RootStack> navigation component, which contains and connects all the other screens in the mobile application together, depending on whether the isLoggedIn state value is true or false.

To change the isLoggedIn state from false to true, the matching set state action is passed down to the <LoginScreen> component as a prop to be used in the authentication flow of the <LoginScreen> component, so that users can login to the mobile application.

### 5.8.2 | BOTTOM TABS

To create an easily accessible tab menu where the user can navigate between the more general screens in the mobile application, I have created the <HomeTabs> navigation component.

The <HomeTabs> component consists of a <Tab.Navigator> component from the React Navigation library, which wraps the following four screen components in our mobile application:<HomeScreen>, <MapScreen>, <MessageScreen>, <FeedScreen>.

Using the <Tab.Screen> component from the React Navigation library, a unique route is assigned to each of the four previously mentioned screen components using the name and component props.

To make the <HomeTabs> component more visually appealing, I have styled it by defining some different screen options in the screenOptions prop of the <Tab.Navigator> component used for the <HomeTabs> component.

```
screenOptions={({ route }) => ({
  tabBarIcon: ({ color, size }) => {
    let icon;

    //Assigns the tap bar icon matching the route name
    switch (route.name) {
      case "Home":
        icon = "home";
        break;
      case "Map":
        icon = "map";
        break;
      case "Messages":
        icon = "chat";
        break;
      case "Feed":
        icon = "post";
        break;
      default:
        break;
    }

    return (
      <MaterialCommunityIcons name={icon} size={size} color={color} />
    );
  },
  headerTintColor: "#fefbd9",
  headerStyle: { backgroundColor: "#1e1c19" },
  tabBarActiveTintColor: "#a6645b",
  tabBarInactiveTintColor: "#fefbd9",
  tabBarStyle: { backgroundColor: "#1e1c19" },
})}
```

Figure 87 | The screenOptions prop of the tab navigator component used for the home tabs component

The styling is applied to each of the different routes in the <HomeTabs> component using an arrow function with a route parameter.

I have made the value of the tabBarIcon property dependent on the return statement of an arrow function with both a color and a size parameter to give each route a unique tab bar icon. The return statement of the arrow function returns an icon using the <MaterialCommunityIcons> component from the Expo Vector Icon library which is made dependent on the three following props: name, color, size.

The values for the color and the size props are given by the corresponding parameters of the arrow function, and the value of the name prop depends on the value of a variable created in the arrow function named icon. To ensure that each route gets a unique tab icon, the route name from the route parameter in the arrow function, which the arrow function that controls the tabBarIcon property is inside, is used as the expression in a switch statement.

In this way the switch statement ensures that the value of the correct unique icon name is assigned to the icon variable, resulting in the arrow function returning the correct icon for each of the routes in the <HomeTabs> component.

### 5.8.2 | NATIVE STACK

I have created the former mentioned <RootStack> navigation component to stack both the <CameraScreen> and the <ChatScreen> components on top of the screens in the <HomeTabs> component of the mobile application.

```
6   const Stack = createNativeStackNavigator();
7
8   export default function RootStack() {
9     return (
10      <Stack.Navigator
11        screenOptions={{
12          headerTintColor: "#fefbd9",
13          headerStyle: { backgroundColor: "#1e1c19" },
14        }}
15      >
16        <Stack.Screen
17          name="Back"
18          component={HomeTabs}
19          options={{ headerShown: false }}
20        />
21
22        <Stack.Screen name="Camera" component={CameraScreen} />
23
24        <Stack.Screen name="Chat" component={ChatScreen} />
25      </Stack.Navigator>
26    );
27  }
```

Figure 88 | The root stack navigation component

The <RootStack> component consists of a <Stack.Navigator> component from the React Navigation library, which wraps the <HomeTabs>, the <CameraScreen> and the <ChatScreen> components.

Using the <Stack.Screen> component from the React Navigation library, a unique route is assigned to each of the three previously mentioned components using the name and component props.

By assigning a route to the <HomeTabs> component in the <RootStack> component, it becomes nested within the <RootStack> component, enabling a connection between the routes assigned in the two navigation components. This connection between the routes assigned in the two navigation components makes it possible to create some navigation between the screens in the two navigation components. In our case it makes it possible to navigate from the home screen to the camera screen and from the message screen to the chat screen in the mobile application.

To make the <RootStack> component more visually appealing, I have styled it by defining some different screen options in the screenOptions prop of the <Stack.Navigator> component used for the <RootStack> component.

Due to the nesting of the <HomeTabs> component in the <RootStack> component, I have also defined the header to be hidden when the app is on one of the screens in the <HomeTabs> component, so the header of the two navigation components does not conflict with each other. To hide the header, I have defined a styling property in the options prop of the <Stack.Screen> component, which assigns a route to the <HomeTabs> component in the <RootStack> component.

## 5.9 | CONTEXTS – VICTOR

To simulate a backend and database functionality for the profile settings, I have created a <ProfileContextProvider> component using both the createContext API and the useState hooks from the React framework.

```
3   export const ProfileContext = createContext(null);
4
5   //Profile context provider component
6   export default function ProfileContextProvider({ children }) {
7     const [profile, setProfile] = useState({
8       username: "",
9       profileImage: "",
10      purposeList: [],
11      location: null,
12      hideLocation: false,
13      interests: [],
14    });
15
16    return (
17      <ProfileContext.Provider value={{ profile, setProfile }}>
18        {children}
19      </ProfileContext.Provider>
20    );
21  }
```

Figure 89 | The profile context provider component

The <ProfileContextProvider> component handles the state of a logged-in user's profile settings in the form of a profile settings object configured using the useState hook from the React framework, which is passed along with the matching set state action in the value prop of the <ProfileContext.Provider> component created using the createContext API from the React framework. The <ProfileContextProvider> is configured to take a children prop, making it a component used to wrap other components, allowing the current state value of the profile settings object to be used or changed at each layer of components inside the

<ProfileContextProvider> component in the mobile application using the useContext hook from the React framework inside the components. This is seen, for example, in the <LocationAddress> component, where it is made possible for the user to show or hide their location using the set action state of the ProfileContext using the useContext hook from the React framework.

```
79                  setProfile({
80                    username: profile.username,
81                    profileImage: profile.profileImage,
82                    purposeList: profile.purposeList,
83                    location: location,
84                    hideLocation: profile.hideLocation === false ? true : false,
85                    interests: profile.interests,
86                  });
```

Figure 90 | The set action state of the ProfileContext used in the location address component to change the profile settings

To ensure that the current state value of the profile settings object can be used or changed at each layer of components inside the mobile application, the <ProfileContextProvider> is used as the parent component in the App.jsx file that contains all the mobile application components.

# 6 | DISCUSSION

Developing FUN!gus presented several significant challenges that ultimately strengthened both our technical skills and collaborative processes. The most substantial obstacle emerged when we needed to establish a new repository due to SDK 53 compatibility issues. This setback initially felt daunting, as it required restructuring our entire codebase and re-establishing our GitHub workflow. However, this challenge became a valuable learning experience that improved our version control practices and taught us the importance of maintaining flexible development environments.

One of the most exciting breakthroughs occurred when implementing our custom component architecture. Creating reusable elements like the IOSButton and TextErrorInput components felt genuinely innovative, as we watched our code become more modular and maintainable. The moment when our filter functionality began working seamlessly across different screens was particularly satisfying. Seeing users being able to search and filter interests in real time demonstrated the power of well-structured React Native development.
Reflecting on our approach, we could have benefited from establishing clearer code review protocols earlier in the process. While our weekly meetings provided good coordination, more frequent peer review sessions might have caught potential issues before they became larger

problems. Additionally, we could have invested more time in comprehensive testing strategies, particularly edge cases in our navigation flow and data handling. These problems are most noticeable when it comes to our filter components. Two different group members had simultaneously developed filters for their respective screens. As it stands currently, our project has two different filter components that solve the same problem with different means. If we had communicated more during the process, we could have developed one filter and wasted less time.

Regarding our initial goals, we successfully created a functional prototype that demonstrates all core features of our intended application. The user can navigate between screens, interact with map markers, engage in conversations, and access educational content about sustainable mushroom cultivation. We are particularly satisfied with how our design vision is translated into working code. The visual consistency and user experience closely match our Figma mockups.

Looking ahead, several improvements would enhance FUN!gus significantly. Implementing real time messaging functionality, adding push notifications for new matches, and developing a proper backend database would transform our prototype into a production and ready application. We would also like to explore advanced geolocation features and integrate actual mushroom cultivation tracking capabilities.

GenAI tools played a supporting role in our development process. Rasmine found Claude particularly valuable for debugging complex filtering logic and as a sparring partner when exploring different architectural approaches. Nina used AI assistance when troubleshooting animation issues in the feed screen overlays, helping identify performance optimization opportunities. Joseph leveraged AI tools for understanding and resolving navigation conflicts. These tools served as knowledgeable coding companions rather than solution providers, helping us think through problems and understand best practices while maintaining ownership of our creative decisions.

# 7 | LITERATURE


Literature

Lupanda, I.S & Rensburg, J.T.J.V (2021) “DESIGN GUIDELINES FOR MOBILE APPLICATIONS” North-West University, South Africa

Links

Beyond Coffee DK (2025) “Dyrk lækre svampe” Retrieved 27.03.25 from:

https://beyondcoffee.dk/?srsltid=AfmBOopsUSKukmLCVWXs1WdOGboUeWzE4m8y5RplgzeI_8AFqKMjp-ij

React Dev (2025) “Built-in React Hooks” (State hook) Retrieved 27.05.25 from:

https://react.dev/reference/react/hooks

# 8 | INDIVIDUAL REFLECTION – NINA

Throughout the three individual assignments submitted during the PMA course, I experienced a significant progression. I have developed both my technical skills using React Native and my ability to reflect on programming as a design practice. Each assignment allowed me to build upon the learnings gained on the previous one and allowed me to explore code as a creative medium and communicative gateway.

In the first assignment, I learned to use core React Native components such as <View>, <Text>, <TextInput>, <Image>, and <Button>. Although the UI was simple, the real challenge was to structure and clean my code. This was to make it readable to my peers, and to teaching staff who would evaluate the assignment. During the show-and-tell, I noticed that some variable names were too personalized and not easy to interpret. This taught me the importance of consistent naming, clear commenting, and thinking about how others should be able to understand my code. Additionally, I also began to reflect on the principles of HCI, especially those about feedback, clarity, and consistency.

The second assignment introduced state, which allowed me to make the application interactive. I implemented toggles for liking a post, switching images, and writing comments. This made my application more dynamic and responsive. I faced challenges in creating a working scrollable comment section, but when I finally solved how the code worked in practice, it helped me understand useState more deeply. Explaining the code to a peer helped me reflect on areas I did not fully understand. I also learned new ways of solving problems by listening to how my peer had approached the assignment. This made me feel more comfortable with my coding skills and motivated me to continue improving.

In the third assignment, I worked with navigation and reusable components. I have structured the application using NavigationContainer and createNativeStackNavigator, which ensures modularity and screen transitions. Creating a reusable <Comment> component helped me to adapt to scalability and separation. I applied HCI principles from lecture 3, such as Norman's concepts of affordances and discoverability, by ensuring the user actions (posting or deleting a comment) had an intuitive outcome.

To summaries, the three assignments helped me develop both technical and reflective skills. I have learned to write more structured and modular code and to integrate user experience principles into my designs. Presenting my code to other peers helped me understand programming as a creative and collaborative design practice. I feel more confident now than before the course.

# 9 | INDIVIDUAL REFLECTION – JOSEPH

While not new to coding, these assignments, and this course in general, have been my first introduction to mobile app development. Prior coding courses taken have almost all been in Python and were taken many years ago - notably programming was never something I much enjoyed or was very good at. While I mostly knew the jargon and my way around an IDE, many things introduced to me throughout the course were as good as new.

The first assignment served mostly as an introduction to styling through stylesheet. Here I explored fitting different elements onto a screen, how to define width and height, both through fixed numbers and percentages. I explored margins and paddings as well as view components and sizing them through flex. I had my first go at an interactive button and felt proud when I managed to get it to show an alert on press. Being required to include images in the project let me upload images of Krtek (a Czech mole cartoon) which became a theme throughout my assignments.

The second assignment let me explore interactivity in mobile applications, something different from my experiences in Python. My favorite experience with this assignment was finally figuring out useState and playing with the changing colours of a like button. While I was able to figure out how to add text into a comment section, my implementation was rather simple, and I regret not spending more time on it. I also forgot to make the user able to change the image on the app, something I would have liked to work with. This assignment was my first experience presenting the code to a fellow student: I felt comfortable using the correct jargon when presenting my code and ideas and equally comfortable understanding my co-student's code.

The comfort dwindled during the third assignment. My pleasant experiences with the previous assignments let me set my expectations high; however, I struggled with most aspects and ended up handing it in late. Structuring components into folders came easily to me, however, wrapping my head around navigation and adding to / taking from lists was a struggle. I sought help from a software developer friend and finally completed the assignment. During presentations I felt comfortable showing my code and could follow the presentation of others despite my early struggles.

Overall, being required to hand in three assignments during the course forced me to get familiar with mobile app development and equipped me well for the final project.

# 10 | INDIVIDUAL REFLECTION – PARNUNA

My experience through the three assignments has been characterized by a growing understanding of what constitutes good code. When I began working on my first assignment, the dice animation project, I had little understanding of how functions should be structured. After receiving feedback from a fellow student about breaking down the rollAnimalDice function, my biggest realization centered around function responsibility. The feedback opened my eyes to the single responsibility principle; each function should have one clear purpose. This was my first understanding that readable code is just as important as functional code. Before this feedback, I had written functions that tried to accomplish multiple tasks simultaneously. The rollAnimalDice function handled input validation, started animations, and selected random animals all within a single block of code. This approach created maintenance nightmares and made debugging significantly more challenging. The realization that functions should focus on doing one thing well altered my coding approach.

Moving forward to my second assignment, the Instagram clone project, my understanding of architecture became deeper. From having everything in one App.js file, I recognized the value of feature-based folder structures and learnt to think in terms of separation of concerns at the entire application level. The styling approach evolved from one large StyleSheet to co-location of styles with components. This project taught me that good code architecture is like good city planning, everything has its place, and related items should be grouped together. I proposed a comprehensive folder structure where components would reside in their own directories with associated styles. The transition from having everything in one place to organizing it systematically was enlightening.

As I progressed to my third assignment, my reflection revealed a new dimension: performance consciousness. Concepts like lazy loading, React.memo, and useCallback became strategic choices for better user experience. Simultaneously, I learnt to accept bugs as learning opportunities, the report functionality's failure became a reminder that development is iterative. This assignment marked my transition from writing code that simply works to writing code that works efficiently. I began considering memory usage and user experience implications of my technical decisions. Perhaps most importantly, I learned to embrace imperfection as a valuable lesson about the iterative nature of coding. Through these three reflections, I have evolved from thinking "does it work?" to "how can it work better?". Most importantly, I have learnt to value code that not only functions but can also be understood and maintained by others.

# 11 | INDIVIDUAL REFLECTION – VICTOR

When programming the mobile application for my first individual assignment, I learned how to use some of the different core components, focusing on how to program a mobile application for an iOS device using the React Native framework. Using a combination of the TouchableOpacity and Text core components, I was able to apply my own styling for buttons, which is not possible when using the Button core component. To ensure that the mobile application's content visually fits the screen of the iOS devices, I used the SafeAreaView core component instead of the View core component as the parent component for the whole mobile application, surrounding all the application components. When I presented the code to one of my peers, I realized that my code was missing written notes explaining the different parts of my code, resulting in a wasted opportunity to document and explain my code, so that it would be easy for others to get an overview and understand the code.

To address the missed opportunity, I found while programming the mobile application for my first individual assignment, I started adding comments while programming the mobile application for my second individual assignment to systemize it for others and myself. This gave my peers a better understanding of my code and helped get rid of the earlier confusion. During the second individual assignment, I also learned about the Pressable core component and the differences between this component and the TouchableOpacity core component. The main difference is that the TouchableOpacity component has a built-in functionality that provides feedback when interacting with it, which the Pressable component does not. However, it is possible to provide feedback when interacting with the Pressable component using the useState hook. I also challenged myself and used different types of states in my application, and in this way, I further learned to push and challenge myself a bit. To achieve a more visually appealing effect, I also used expo vector icons for this individual assignment.

When I was programming the mobile application for my third and final individual assignment, I learned about the React Navigation library and how to use the Native Stack solution to create different screens using screen components and navigate between them. In addition, I also learned about how to create custom components and use them to break down the entire application into both reusable and structural components. Finally, when I presented the code to one of my peers, I quickly found out that I had created a good structure for my code, so it was easy to get an overview, understand and explain. The structure of my code consisted of a folder system of two folders. Each of the two folders in the folder system has its own content, one for the different screens in the mobile application and one for the different components. The structure of the code in my third individual assignment is something I would like to take with me for the rest of my course, as it has a great advantage in providing an overview and explaining my code, just like writing comments.