



DANMARKS TEKNISKE UNIVERSITET

62550 UX MOBILE APP DEVELOPMENT

GROUP KD1

Project Be Kind

Authors

Mads SØRENSEN

Mark NIELSEN

Lucas SCHOUBYE

August HJORTHOLM

Espen ULFF-MØLLER

Course responsible

Ian bridgwood



Mads Sørensen
s215805



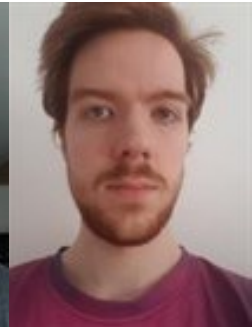
Mark Nielsen
s204434



Lucas Schoubye
s215801



Espen Ulff-Møller
s215831



August Hjortholm
s215800

January 2023

Link to Github: [Here](#)¹

Link to Figma: [Here](#)²

Login details for the app (prefilled when launching app):

Email: kd1@kind.dk

Password: 123456

¹<https://github.com/MadSoeDK/Kind>

²<https://www.figma.com/file/GWBlvzZ4W9eH13HGiqvyGy/Be-kind-projekt?node-id=1066%3A3966t=UF2tS1rESBQZYwuZ-1>

Contents

1	Introduction	3
1.1	Vision	3
1.2	User manual	3
1.3	General requirements	3
1.4	Group member contributions & Timetables	3
1.5	Group conflict	3
2	Requirements	4
3	Analysis	6
3.1	Domain Model	6
3.2	User stories	6
4	Design	8
4.1	UX Elements	8
4.2	Dark mode	9
4.3	Class diagram	9
4.4	Activity diagrams	11
4.5	Sequence diagram	12
4.6	Navigation diagram	14
4.7	Architecture	14
5	Implementation	16
5.1	Firebase	16
5.2	Navigation	17
5.3	UI implementation & Composables	18
5.4	ViewModel	19
5.5	Data layer	19
5.6	Model	19
5.6.1	Service implementation & Firebase	20
5.7	Stripe integration	20
5.8	Theme	21
5.8.1	Darkmode	21
6	Testing	23
6.1	User test	23
6.2	Internal testing	23
6.3	Acceptance	23
6.4	Unit test	23
6.5	Known issues	23
7	Conclusion	24
7.1	Future Developments	24
A	User manual	25
B	Time Spent	29
C	Composable Function implementation example	32
D	Viewmodel implementation example	32
E	Payment intent object	33

F Formal Tests

34

1 Introduction

In this project, we were paired with the company Kind, that wants to make donations to charities easier, more transparent and accessible for everyone. We were tasked with implementing an app that could fulfill this vision. This report documents the analysis, design, implementation and testing of the app.

1.1 Vision

For anyone who wishes to donate to charity, Kind is a donation platform, that wants an easy and transparent solution for all donations. Kind seeks to make it easy and transparent for the user to find the right organisations to donate to unlike traditional ways of donating. Our Product makes the process of donating to a portfolio of charities easy and transparent for the user.

1.2 User manual

A guide on how to navigate the app can be found in appendix A

1.3 General requirements

The product owner (PO) wished you are able to donate to charities and build a portfolio. It was intended that charities could have their own accounts and be able to release articles.

1.4 Group member contributions & Timetables

The timetable and descriptive texts about group contribution can be found in appendix B.

The number of commits found under "Insights" in the Github repo is not a standalone metric of group contribution, but can be compared with the timetable to get a fair insight in each members contribution. The overall for each group member is summed to:

August: 112

Espen: 100

Mads: 190

Lucas: 145

Mark 150

1.5 Group conflict

The group contract was breached several different times by group member (s215800). The issues were primarily a lack of communication, not attending agreed meetings and working days, and was overall unreliable to finish agreed work on time. We encountered multiple days where the member would fail to show up, and could not be reached or contacted, often excused by oversleeping the whole day, and therefore unable to be present at the most important phases of the project. Overall his contribution to the project was disappointing and lackluster.

2 Requirements

The requirements are developed and prioritized in cooperation with the project owner. The requirement list below is the final iteration in the agile development process and is refined multiple times throughout the project. MosCow's model is used for prioritization. The requirement for our MVP would therefore be the requirements found in our "Must Have" category.

ID	Functional requirements
Must have	
M01	The user must be able to donate to charities.
M02	The user must be able to learn about different charities and their cause.
M03	The user must be able to follow a charity, and see articles related to the charity.
M04	The user must be able to see how their donations are distributed.
M05	The user must be able to create a profile.
M06	The user must be able to add and remove charities from their portfolio.
M07	The user must be able to reset their password.
M08	The user must be able to login to their account.

ID	Functional requirements
Should have	
S09	There should be a feed with news about relevant updates of the charity.
S10	The user should be able to transfer funds to a charity using stripe.
S11	The user should be able to create a subscription which donates monthly to the chosen organizations.
S12	The user should be able to edit their subscription to change the organizations donated to and the amount.
S13	The user should be able to see a list of all their donations made.
S14	The user should be able to edit their account information, such as email and password.

ID	Functional requirements
Could have	
C15	The user should be able to navigate to a home page which shows articles of the charities being followed.
C16	The user should be able to log in with Google.
C17	The user should be able to navigate to a explore page, which shows charities who are not followed.
C18	The user should be able to navigate to a portfolio page, which displays detailed information about how the user spending.
C19	The user should be able to navigate to a charity, where they can see details about the charity and see articles posted by the charity.
C20	The user should be able to delete their account.

ID	Functional requirements
Won't have	
W21	The user should be able to see a message saying "you are in the top x contributors".
W22	The user should be able to see a message saying "you donate top x".

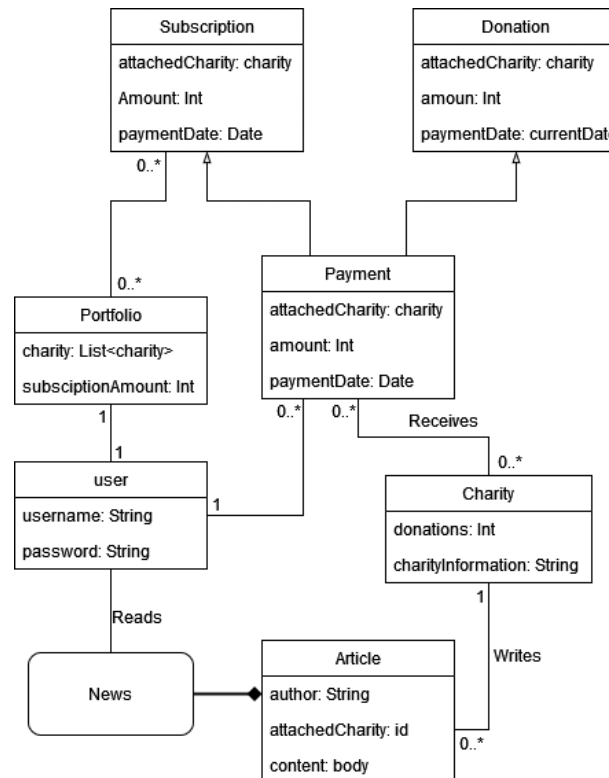
ID	Non-Functional requirements
Non-Functional	
N1	The user should see a UI response when waiting for the UI to load.
N2	The app development process should be scalable in the future.

3 Analysis

In this section we will document the analysis we did for this project. This will include what models we used and the user stories we created.

3.1 Domain Model

Our domain model is shown below, which provides conceptual modeling of how our behavior and data will be organized.



Each user has a list of subscriptions that represents the portfolio. A subscription is a recurring donation. A donation is There are also multiple charities. Each charity has a theme such as "Children's Welfare". These charities can create articles that the user will read in their news box.

3.2 User stories

We have defined some core user stories. For the user stories, we have used the word 'User' for user type. A more descriptive word for our user could have been "donor", but we decided to go with "User" as the vision for the app is for everyone to be able to explore charities without having to donate. The user stories is defined in cooperation with the project owner.

- **Build portfolio**

As A user

I would like to be able to build a portfolio of monthly automatic donations to charities so I can manage my donations more easily and cheaper.

- **Edit portfolio**

As a user

I would like to be able to edit my monthly donation amount and the selected charities

as this would allow me to customize donations to charities from month to month.

- **One-time donation**

As a user

I would like to be able to choose a charity and donate a one-time amount,

as this gives me the ability to quickly donate to a cause without monthly subscriptions

- **Subscribe to charities**

As a user

I would like to be able to make a monthly-based transaction to a charity,

as this automates my billing instead of doing it manually.

- **Explore charities**

As a user

I would like to be able to explore and read more about charities,

as this gives a better overview of whether it is a charity I am interested in supporting.

- **Read charity news**

As a user

I would like to be able to read news from my charities,

as this gives the motivation to keep supporting.

- **Read charity news**

As a user

I would like to be able to read news from my charities,

as this gives the motivation to keep supporting.

- **Sign up**

As a user

I would like to be able to make an account,

as this ensures I can keep my data saved and stored.

- **Login**

As a user

I would like to be able to log in to my account,

as this gives me access to my stored data.

- **View billing history**

As a user

I would like to be able to view a history of my transactions,

as this gives me an organized view of my payments.

4 Design

In this section, we will go over our design decisions for our app. This will primarily be shown with Figma diagrams and explanatory text. We will also discuss the structure of our application's code.

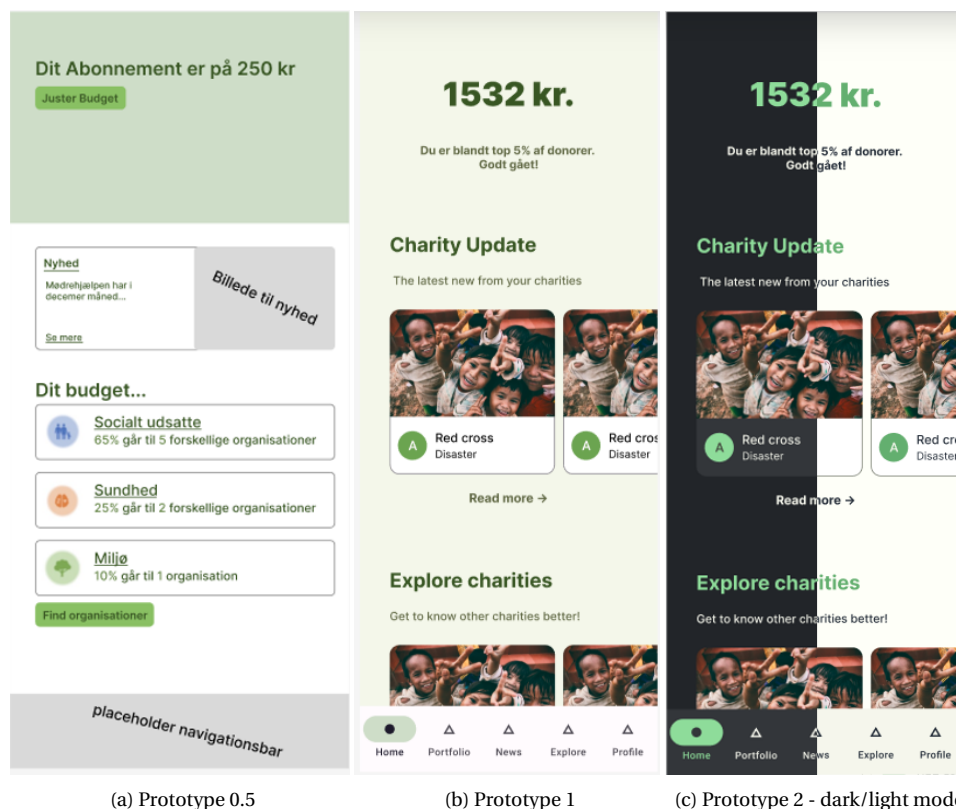
4.1 UX Elements

We have made certain general design decisions with the Kind "brand" and vision in mind. A charitable app should bring a "feel-good" vibe to the user, and this is reflected in the app. The green color was chosen, and the buttons are rounded because it gives off a good and positive feeling when the user looks at it.

We have made custom designs of our app components according to universal design principles and accessibility design guidelines. This includes buttons, cards, TextFields etc. For accessibility we have set contentDescriptions and clickLabels on our composables.

Prototype 0.5 to 1, had a complete overhaul and redesign, with only the base concepts and colors being kept. In prototype 1 we arrived at a great flow, which is implemented in Figma. Prototype 2 was mainly modernized to full usage of material design and colors, the flow of the app was kept from iteration 1.

Link to Figma prototype design: [Figma](#). The different boards and iteration can be selected in the topleft corner in Figma.



We have chosen to show both parts of prototype 2, as we have both a dark and a light theme.

Our design builds on the concept of the good feeling after a charitable donation. As shown in the differences between prototype 0.5 and 1, this came to mind.

Through the iterations, we enlarged the total donated sum amount on the HomeScreen, framing it more like an achievement and it is the first thing you should see when launching the app

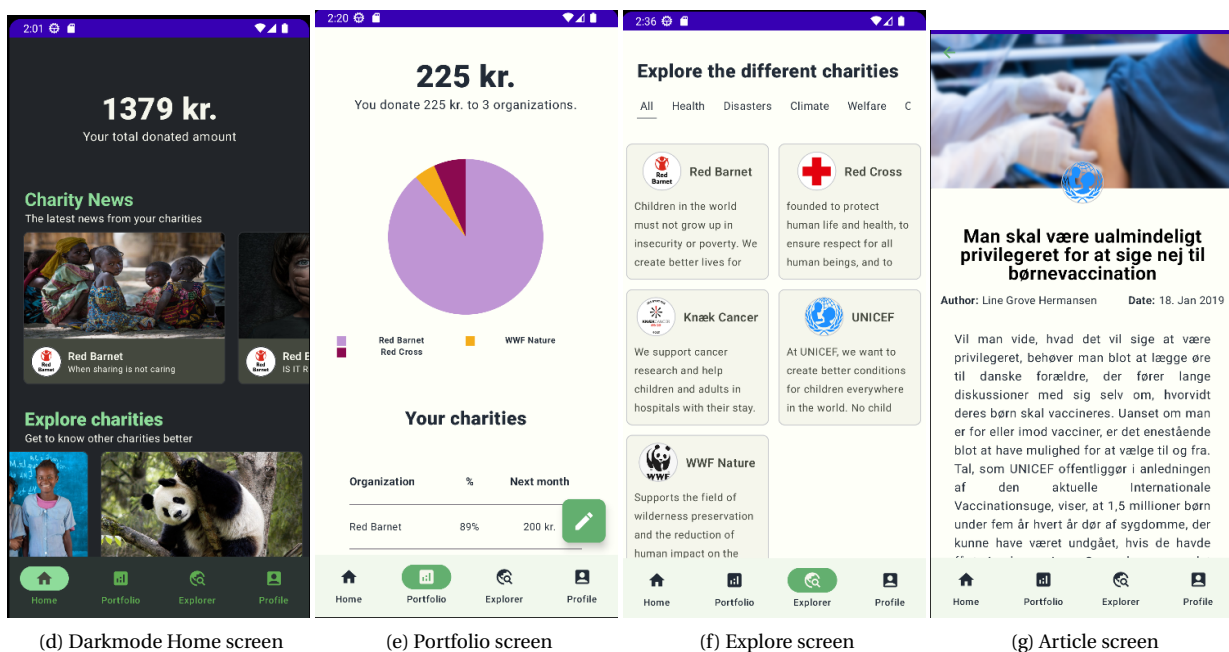
We kept the navigation at the bottom, where most users will hold their phones, close to the users' thumbs, and arranging the important elements close to the bottom, but visible from the launch of the screen. This is universal design and is also very similar to many other apps, so the user will know how to navigate when launching our app.

Gestalt design principles have been used. We reduced the number of elements visible on most screens, and used spacing to separate sections with the Law of Proximity, eg. the "Charity Update" and "Explore Charities" on the homepage. We kept elements grouped via horizontal scrollable sections, which works great due to the Law of Common Region and Similarity.

Another UX principles our app fulfills is the Doherty Threshold ³, which states that "productivity soars when a computer and its users interact at a pace that ensures that neither has to wait on the other". The app runs quick and smoothly, and just in case the network is slow we have added loading screens to tell the user that something is happening in the background.

4.2 Dark mode

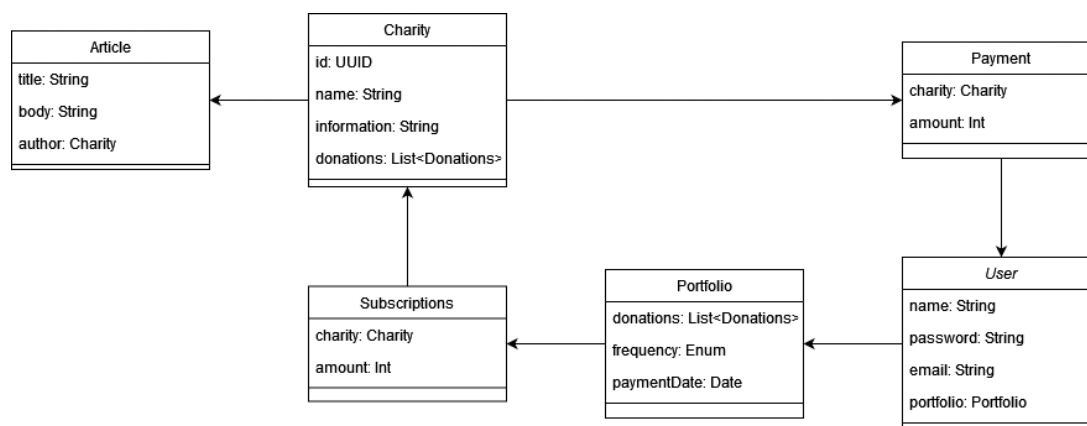
We decided to include a dark mode for our users. There are plenty of benefits to this. Besides just having a different look to the app, it can increase the battery life of the hosting device as well as produce less blue light, improving the user experience at late hours.



4.3 Class diagram

We have made the following class diagram of the model part in our MVVM, Model-View-ViewModel, architecture. This differs from the domain model, as we have abstracted some concepts away, as they do not hold much core functionality. They basically map our data access objects used for requesting data and sending data to and from Firebase.

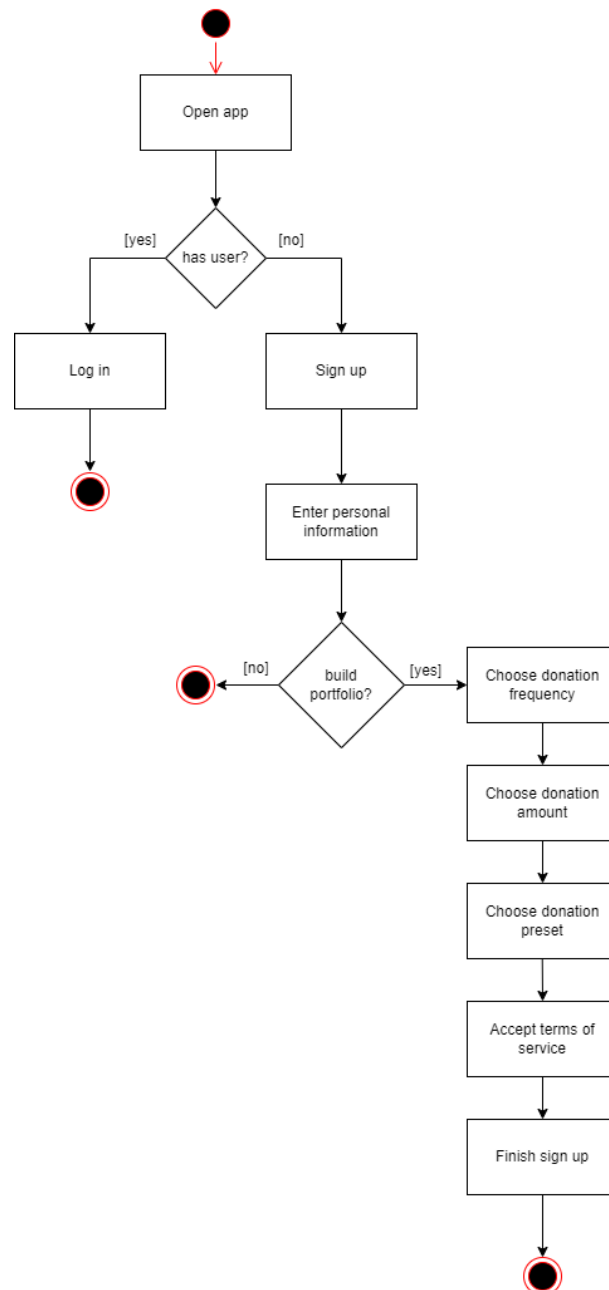
³lawsofux.com/doherty-threshold/



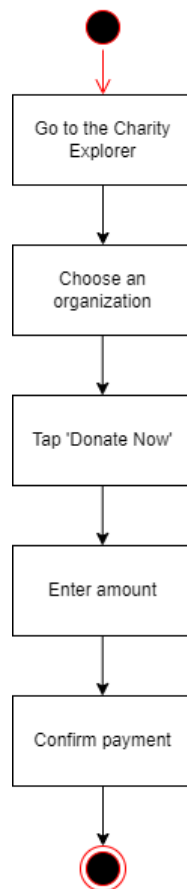
As evident from the diagram there are no standalone functions on the model, as these are handled by the StorageService and AccountService..

4.4 Activity diagrams

We have made activity diagrams for some of the user stories. Below is the activity diagram of the sign-up flow and the build portfolio user story. The user could potentially have a long sign-up flow, it's almost a questionnaire for getting enough information to start a portfolio. It's important to onboard users to the portfolio of charities. We give the user the option to leave the signup flow earlier, and not have a portfolio in the beginning, to not lose users in the signup process. The app does not require a subscription to use it.

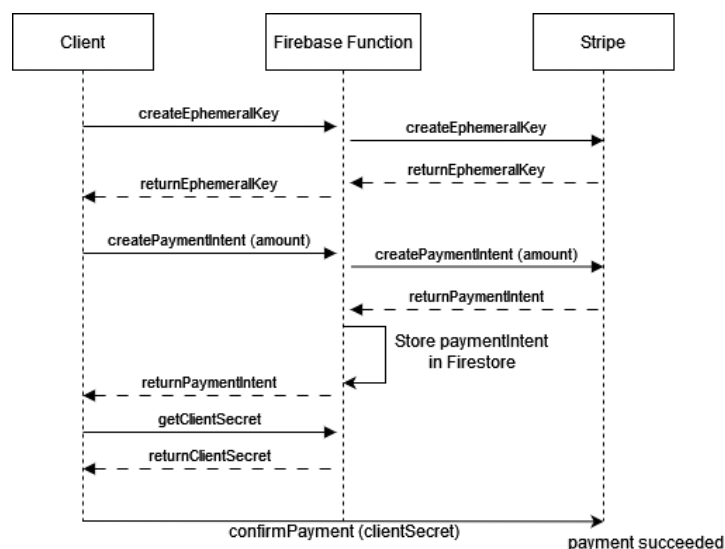


Next, is the activity diagram over a one-time donation. It assumes that the user has already created an account and is logged in. Note that the payment processing section done by Stripe is not included in the diagram.

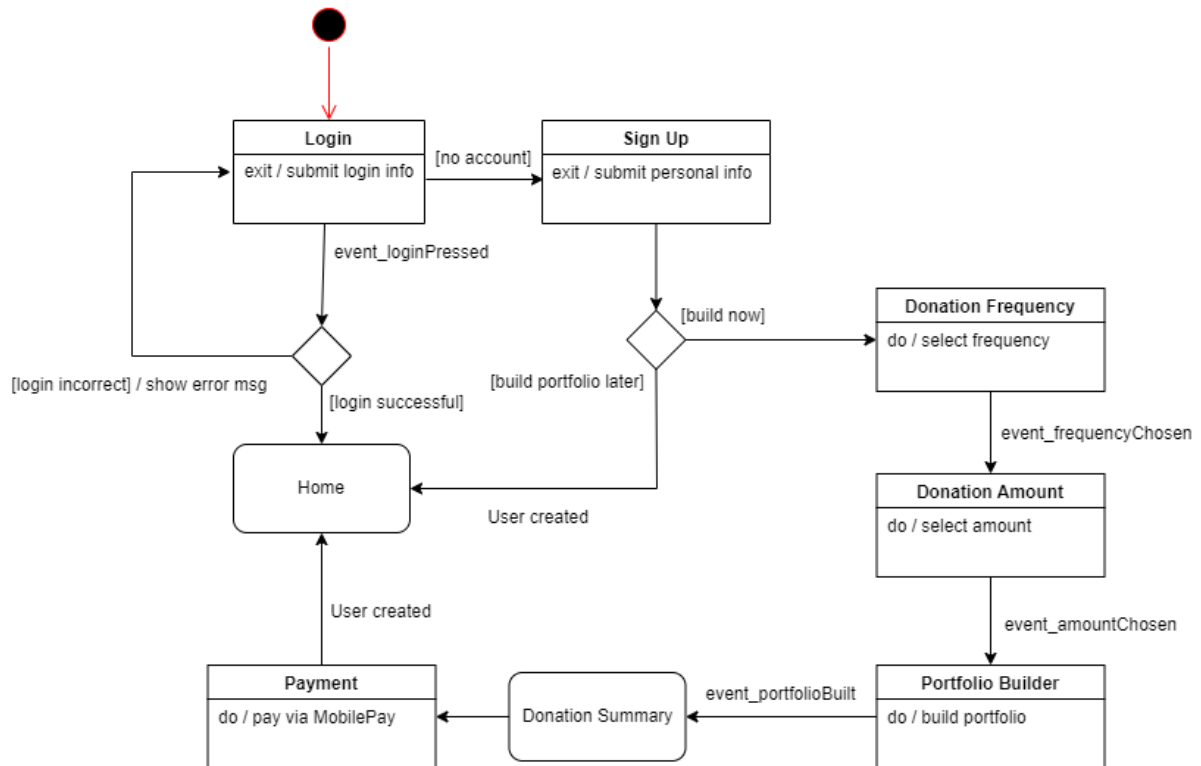


4.5 Sequence diagram

The sequence diagram shows we integrate payment with Stripe. First, the user clicks the 'Donate' button, which request an EphemeralKey, and returns it. Then the user enters the amount to be donated and selects the payment method. That requests a PaymentIntent to Stripe, and an object is returned and stored in Firestore. Lastly to confirm the payment, the client fetches the client_secret, now stored in Firestore, and confirms the payment to Stripe with the client_secret.

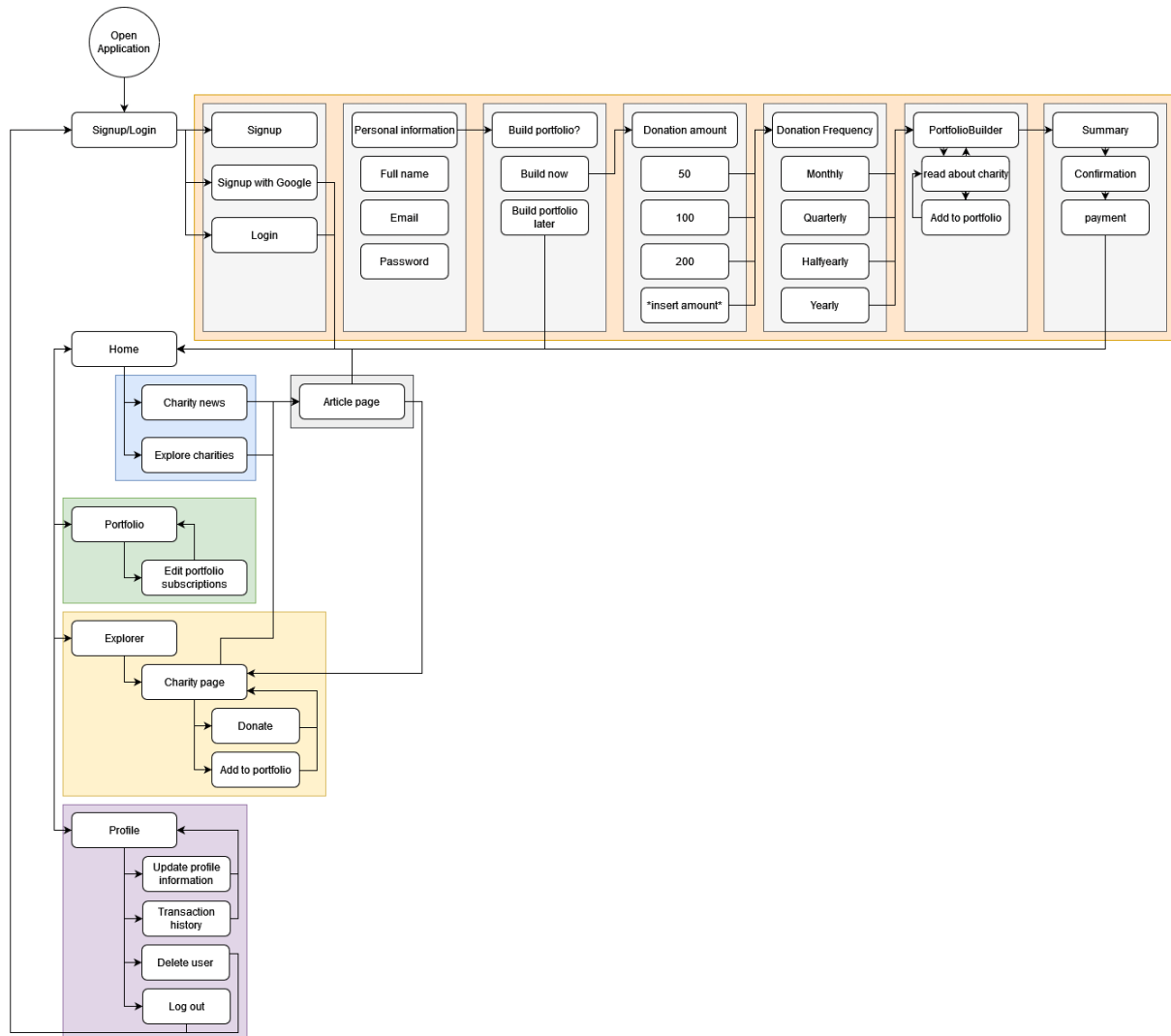


Below is shown a state diagram of when a user has to access the application with an account, where they can log in if they already have a user or sign up if they don't.



4.6 Navigation diagram

Below you can see our navigation diagram. At the top, you can see our login flow. You can see that when logging in you are only able to access the rest of the app in a few ways. These include 'login', 'sign-in with google', 'build portfolio later', and entering their account information. Hereafter we have entered the main functionality of the app. We have a few main screens, these include the home screen, explore screen, portfolio screen, article screen, and profile screen.



4.7 Architecture

We have used a few different design and architecture patterns to maintain a clean and structured project.

- **Model-View-ViewModel (MVVM):** We have organized our code into 3 layers and each has their responsibility. The Model takes care of abstracting the data source(Firebase, Stripe etc.) to use in the program. View informs the ViewModel of the user's actions and observes if the data in ViewModel is changing. ViewModel is linked with both Model and View and exposes the data to the View, but also works together with Model to get and save data.
- **Observer pattern:** The Jetpack Compose framework we use for UI has the Observer pattern built into it. This means a lot is handled by the framework, but we as developers have to tell which data is observed by the view, and when that data changes the view does recomposition on the components that depends on that data.

- **Repository pattern:** Repository patterns encapsulates the logic to access the data source to abstract it from the rest of the app. We have 2 repositories, one for our database in Firestore, and another for user authentication in FirebaseAuth. The repositories would implement an interface of our usecases, and if we decided to swap the backend, we would just make a new implementation of that interface, and the rest of the app stays the same.
- **RESTful API:** We need to handle some data on the server such as user information, payment handling, user generated content (charity articles for example) etc. The communication with the REST API would be via HTTP using the Firebase and Stripe SDKs.
- **Singleton Pattern** The singleton pattern limits the creation of a class to only one object. This can be beneficial if a system uses an object to coordinate between systems. In order to make a object a singleton, the constructor have to be private, which means no object, other than itself can access the constructor. The service implementations, StorageService and AccountService are Singletons, we only need a single object of them across the application.

Figure 6 shows our app architecture described from the patterns above.

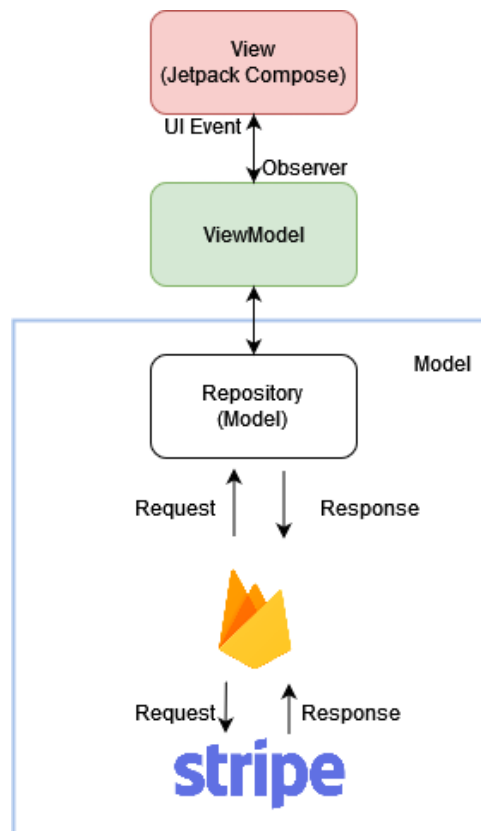


Figure 6: App Architecture

5 Implementation

Code snippets provided in the implementation can be simplified to improve readability. See the Github for full code. The project is structured by the MVVM architecture

Kind

/firebasefunctions: The firebase cloud functions to integrate the app with Stripe

/app/...

/model

/service : FirebaseAuth and FirebaseFirestore implementation

model.kt : All data classes

/view: The view screens are structured by the navigation tree

/auth_screens

/main_screens

/signup_screens

/composable

/theme

/viewModel

MainActivity.kt

Kindapp.kt

Firebase is used as the back end. This includes Firestore for the database for data storage, Firebase authentication for user authentication, and Firebase functions used to integrate our Stripe payment functionality. The app in Jetpack Compose and the backend is in Kotlin. The Firebase functions are developed with JavaScript and we used the code from the Stripe documentation⁴ and the associated Stripe example Github repository⁵. We will not document the Firebase function code in this report, as it is out of scope for the course, and we have just used the functions as a "black box" to integrate our UI and client with Stripe. We will however document how the client app integrates Stripe and Firebase functions.

5.1 Firebase

The Firestore database is divided into 4 main collections: Users, Articles, Charities, and Customers.

- **User** consists of a UID, a name, a monthly payment, and a payment method. Email, password, and UID are stored in FirebaseAuth and share the same UID so we can look up the same user in two different places. The user has two sub-collections, a donation collection, and a subscription collection.
 - **Donation** collection is used to store the user's donations and stores the amount donated, the charity to which the donation was made, and the date of the donation.
 - **Subscription** document has the same values, but the main difference between the two collections is that a donation is treated as a transaction, while the subscription collection is treated as a timer for the next donation, i.e. once a month a new donation is created for each subscription with the respective amounts and charities as values. So to sum up a Subscription is additional data associated with a donation.
- **Charity** consists of an ID, a category, the number of donors, the donation amount, URLs to the images used for the charities, and a description. A charity also has a subcollection of article references. This way, we assign articles that are relevant to a charity by giving the charity a reference to the article.

⁴<https://stripe.com/docs/mobile/android/>

⁵<https://github.com/stripe-archive/firebase-mobile-payments> JavaScript

- **Article** has a title, text, URLs to the images used in the articles, and the name of the charity associated with it
- **Customer** has a UID the same as the user, so we can reference it correctly, and a customer id which references the customer object in Stripe. It has a Payment subcollection with payment data
 - **Payment**: consists of id, amount, date, and additional data important to the transaction such as client_secret and ephemeral_key.

5.2 Navigation

We have used the Jetpack Compose Navigation library for navigating between different composable. We have a nested navigation tree to modularize our app and keep the app architecture organized.

NavHost

- HomeNavGraph
- AuthNavGraph
- SignupNavGraph

The navigation can be illustrated by the figure below

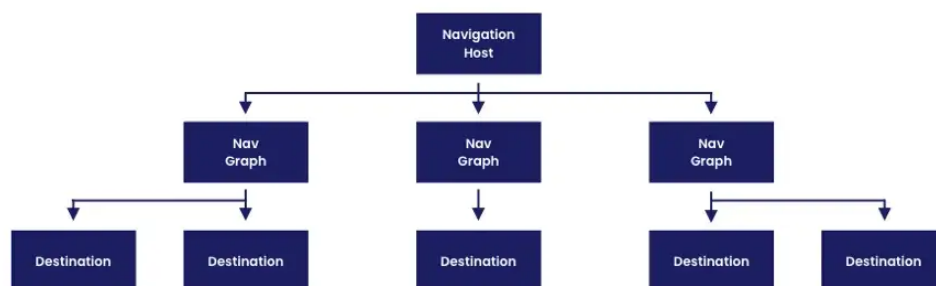


Figure 7: Navigation structure illustration⁶

The different routes are defined in the following way

```

1 sealed class SignupScreens(val route: String) {
2     object Root : SignupScreens("signup_root")
3     ...
4 }
5 sealed class AuthenticationScreens(val route: String) {
6     object Root : AuthenticationScreens("auth_root")
7     ...
8 }
9 sealed class HomeScreens(val route: String, var icon: ImageVector) {
10     object Root : HomeScreens("root", Icons.Filled.Favorite)
11     ...
12 }
  
```

The navigation tree is built with the NavHost as root. We define the startdestination based on whether the user is logged in or not to allow dynamic login/logout functionality.

⁶<https://hitherejoe.medium.com/nested-navigation-graphs-in-jetpack-compose-dc0ada1d4726>

```

1 NavHost(
2     navController = navController,
3     startDestination = if (loggedIn) HomeScreens.Root.route else AuthenticationScreens.Root.route
4 ) {
5     homeNavGraph(...)
6     authNavGraph(...)
7     signupNavGraph(...)
8 }

```

The navigation graph subtrees are created with the `navGraphBuilder` and we define the Root route as well as each route for every composable.

```

1 fun NavGraphBuilder.signupNavGraph(...) {
2     navigation(
3         startDestination = SignupScreens.Signup.route,
4         route = SignupScreens.Root.route
5     ) {
6         composable(route = SignupScreens.Signup.route) {
7             Screen() {...}
8         }
9     }
10 }

```

For some screens, we also navigate with arguments. For example, the Charity screen is defined as "charity/{id}". This allows us to navigate to unlimited uniquely defined Charity screens, and we can fill the contents of each unique screen with the specific charity contents.

```

1 composable (
2     route = "charity/{id}",
3     arguments = listOf(navArgument("id") { type = NavType.StringType })
4 ) {
5     Screen(it.arguments?.getString("id"))
6 }

```

The project has a complex structure and therefore implements our navigation architecture, which would allow us to better scale the app in the future.

5.3 UI implementation & Composables

The UI is created with composable functions and allows the app to have a dynamic user interface by observing the state and utilizing recomposition when a state changes. The principle of state hoisting has been used to move the state into the upper layers of the composable functions and have stateless composable. This makes the composable more reusable, modular, maintainable, and loosely coupled to the rest of the code. By moving the state we ensure a single source of truth to be shared among multiple composables.

An example of such composable can be found in appendix C

For many of our composables we have implemented a hierarchy of composables that "inherits" the properties by nesting them. For example we have an topmost Screen composable defined, that all particular screens use for maintaining the scaffold state, vertical scroll etc. We can then pass in other optional composables functions, such as the floating action button bar, navigation bar etc. to the Screen composable. This makes it very flexible and reusable everywhere in our app.

```

1 fun Screen (
2     modifier: Modifier = Modifier,
3     NavigationBar: @Composable () -> Unit = {},
4     FloatingActionButton: @Composable () -> Unit = {},
5     signupNavigation: @Composable () -> Unit = {},
6     content: @Composable (PaddingValues) -> Unit
7 ) {
8     Scaffold (

```

```

9     bottomBar = NavigationBar,
10     floatingActionButton = FloatingActionButton
11 ) {
12     Column(
13         modifier = modifier
14         .padding(it)
15         .verticalScroll(rememberScrollState()),
16         horizontalAlignment = Alignment.CenterHorizontally
17     ) {
18         content(it)
19         signupNavigation()
20     }
21 }
22 }

```

5.4 ViewModel

ViewModel is one of the architecture components implemented in the app. In general, the app has one View-Model for each screen, but multiple screens can also share the same view model as in the sign-up flow. The main purpose is to hold and update the state of the composable and exchange data with the model layer.

State can be defined in many different ways, but one common way is to use `MutableStateFlow` in the ViewModel which creates a state-holder observable.

```

1 private val _data = MutableStateFlow<Article>()
2 val data: StateFlow<Article> = _data.asStateFlow()

```

To update the state and therefore trigger recomposition the `.update` method is used in the view model by the fetched data from the data layer.

```

1 _data.update {
2     storage.getArticles()
3 }

```

The state is collected on the ui by calling `.collectAsState()`

```

1 val state by viewModel.data.collectAsState()

```

And the variable state can then be passed to composables which are recomposed on state change.

An example of a fully implemented viewModel can be found in appendix D

5.5 Data layer

The data layer consists of 2 repositories, `AccountService` and `StorageService` that expose the data to the rest of the app.

5.6 Model

From the domain model and class diagram, we have defined the data classes to hold the domain data modeled in earlier chapters. The data class below shows the fields related to a Charity.

```

1 data class Charity (
2     var donaters: Int = 0,
3     var donations: Int = 0,
4     val id: String = "",
5     val desc: String = "Check out this charity with the button below",
6     val iconImage: String = "",
7     val mainImage: String = "",
8     var name: String = "",
9     val articles: List<Article> = listOf(),

```

```

10     val inPortfolio: Boolean = false ,
11     val category: CharityCategory? = null
12 )

```

We use the models in the app to represent our state and to represent our data in the database.

5.6.1 Service implementation & Firebase

Firestore is the data source for our StorageService repository. We have defined a number of methods in the interface that corresponds to the basic use cases of our app such as

```

1 suspend fun getSubscriptions() : List<Subscription>
2 suspend fun createStripePaymentIntent(amount: Double, currency: String, charityId: String)
3 suspend fun getCharities() : List<Charity>
4 ...

```

The implementation of the interface, StorageServiceImpl, requests Firebase and returns the result to the app. The keyword "suspend" is used to asynchronously request the data in another thread using Coroutines which block the UI and will make the app feel more responsive. The Coroutinescope is handled in the ViewModel using viewmodelscope.launch Firebase handles the deserialization with the .toObjects() method. An example can be seen below.

```

1 override suspend fun getArticle(id: String): Article? {
2     return Firebase.firestore.collection("Articles").document(id).get().await().toObject()
3 }

```

We also use FirebaseAuth as the data source for the AccountService repository, which represents the data source for users. The FirebaseAuth SDK handles many of the requests, and we just have to provide the parameters for the method. The structure is the same as the previous one.

```

1 override suspend fun authenticateUser(email: String, password: String) {
2     Firebase.auth.signInWithEmailAndPassword(email, password).await()
3 }

```

This sets the currentUser object that we can use for other methods that use the currentUser object to fetch user-related data.

5.7 Stripe integration

Stripe is a payment gateway that allows the app to receive payments from common online payment methods such as Visa, Mastercard, etc. We developed this feature in the app based on the Stripe documentation⁷ and the associated Stripe example Github repository⁸. One issue we had with the integration is that Stripe SDK does not yet fully support Jetpack Compose yet, so we had to refactor some of the "older" code examples in the documentation to fit in this project. First, we create an account on Stripe and set up a private API key in the app. Next, we set up Firebase functions to run server code that integrates Stripe to securely key-exchange EphemeralKey and secret_keys. As stated earlier we do not go into details with the Firebase functions, but the main purpose is to integrate Stripe and save payment data into our database.

When a user navigates to the payment screen, the client creates a customer session, using an ephemeralKey with a firebase function to securely exchange card information.

```

1 CustomerSession.initCustomerSession(context, ephemeralKeyProvider = FirebaseEphemeralKeyProvider())

```

When the user has entered the amount to be paid to a charity and presses the payment method button, it requests a PaymentIntent object from Stripe and saves it to Firestore. See appendix E for the contents of the object. Most importantly the PaymentIntent object contains the "secret_key" used to complete the payment. The code below shows how the client fetches the secret_key.

⁷<https://stripe.com/docs/mobile/android/>

⁸<https://github.com/stripe-archive/firebase-mobile-payments>

```

1 val payment = Firebase.firestore.collection("stripe_customers")
2     .document(Firebase.auth.currentUser?.uid.toString())
3     .collection("payments")
4     .document(doc.id).get().await().toObject<KindPaymentIntent>()!!
5
6 return payment.client_secret

```

We can now launch Stripe UI to present the user with entering their card information. We can use a test card to test a payment. Use 4242 4242 4242 4242 with a date in the future, and a random 3-digit CCV number.

```

1 paymentSession?.presentPaymentMethodSelection()

```

And lastly, confirm the payment when the user presses "Pay" with the client_secret.

```

1 KindButton(
2     onClick = {
3         viewModel.paymentPending = true
4         viewModel.paymentSuccess = false
5         if (paymentState.paymentMethod == null)
6             return@KindButton //Error
7
8         paymentLauncher.confirm(
9             ConfirmPaymentIntentParams.createWithPaymentMethodId(
10                 paymentMethodId = paymentState.paymentMethod?.id!!,
11                 clientSecret = paymentState.clientSecret!!
12             )
13         )
14     },
15     textProvider = "Pay"
16 )

```

And thus, if everything goes well, the payment is confirmed on the Stripe account. The files relating to payment are PaymentScreen, PaymentViewModel, and FirebaseEmphemeralKeyProvider and the backend code is in the firebasefunctions folder.

5.8 Theme

After making a rough draft, we used the "MaterialTheme generator tool" to generate a general theme template from our basic colors. The template came written in Kotlin with declaration names, which works in tandem with MaterialTheme3. The colors are specified in a separate file with only colors.

```

1 private val LightColors = lightColorScheme(
2     primary = md_theme_light_primary,
3     onPrimary = md_theme_light_onPrimary,
4     ...
5 )

```

Here is how the colors are defined for the app. The colors can be called with "MaterialTheme.colorscheme.primary" as an example. This call has been used to control where to have what color if found necessary, while also being abstract enough to keep the scheme in line while in both light- or dark mode.

We have also used this method for font styles, typography, and shapes, as shown here:

```

1 MaterialTheme(
2     colorScheme = colors,
3     typography = Typography,
4     shapes = Shapes,
5     ...
6 )

```

5.8.1 Darkmode

We wanted a simple way to implement a dark theme, also called dark mode. This could easily be done the following way

```
1 val colors = if (!useDarkTheme) {  
2     LightColors  
3 } else {  
4     DarkColors  
5 }
```

We found however that we wished for some differences between our light- and dark mode. There were some specific places, where we wanted a specific color outside of what MaterialTheme had automated via its colorscheme. We spend a lot of time to fine tune the colors in all places of the app.

```
1 containerColor = if (isSystemInDarkTheme() ) {  
2     if (it == selectedOption) MaterialTheme.colorScheme.secondaryContainer  
3     else MaterialTheme.colorScheme.outline  
4 } else {  
5 if (it == selectedOption) MaterialTheme.colorScheme.primary  
6     else MaterialTheme.colorScheme.secondaryContainer  
7 }
```

6 Testing

In this section, we will document how we tested the requirements and if we have fulfilled them.

6.1 User test

We asked non-technical persons and the project provider to test the app and give feedback on our design and functionality throughout the project. We requested that they tried to navigate the flow of our Figma prototype. They were asked to 'speak out loud' and comment on anything. We were given feedback on our color palette and our layout. This feedback helped us greatly to improve the design and appereance of our app and the design section clearly shows the improvement.

6.2 Internal testing

A majority of our testing was done internally. We designed, implemented, and tested our features by agile principles in sprints. The tests were done by using the emulator or many of us had Android devices, so we could test it physically. Each feature was documented in a KanBan board and the state of the testing was organized as such. This has led to a very robust app. this will be documented further in the conclusion.

6.3 Acceptance

We conducted acceptance tests. These are documented in appendix F. We used this data to make sure we have could conclude the requirements necessary for our main use cases were implemented and functional. //

6.4 Unit test

We tried to implement Unit testing. There have been a lot of incompatibility issues. Unit testing in kotlin coroutines requires run blocking. But if these unit tests require information from Firestore they crash, simply because it is not supported. We asked for assistance from the teaching assistants who also were unable to solve this issue.

6.5 Known issues

Our testing has also found some known issues we haven't had the time to correct.

- **Stripe loading:** is unreliable especially with a bad internet connection the transaction is not able to finish, before the UI has continued. This leads to unexpected behaviour and sometimes unhandled.
- **Double loading on subpages:** This is due to the navController being passed around to composable screens that are recomposed, and therefore the navController refreshes. This fix is a bit complex, and we could therefore not make it.
- **Update user information:** User account information is not updated correctly. We did not have time to debug this issue.

7 Conclusion

We have designed and developed a functional app for Kind with the most important requirements implemented.

All the 'Must have' requirements (M01-M05) are implemented. This means that we have covered the requirements for the minimum viable product. We can conclude that we have reached all 'should have' requirements except S11. The product is fully able to create a new subscription in the Firestore database and edit that subscription. Users are able to create one-time donations through Stripe. We did however not integrate the subscription with Stripe, due to time constraints. The stripe API is also outdated for the use of composable and would require more time reverse engineer.

We can also conclude that we have reached all but one 'Could have' requirement (C16). The requirement would allow the user to sign up directly via. google account using OAuth.

We had a great chemistry with the PO, with frequent physical meetings and lots of feedback that we were able to implement. This had a great influence on the project.

7.1 Future Developments

Every requirements was not met and the project was also scoped accordingly. In this section, we will topics of future development.

- Integrate with Stripe subscriptions that would allow users to donate to a portfolio of charities
- Provide offline support and better handle situations with no internet.
- Performance improvements and scalability for example implement paging that would allow to dynamically fetch information as the user scrolls. If a charity for example has hundreds of articles this would take a long time to fetch from the database at the same time. A bit like lazyloading but for Firebase.
- We would also like charities would be able to run charity "campaigns". A user would then be able to donate directly to a time-limited campaign. An example of this could be "Red Barnet" doing a campaign for children of Ukraine.
- Multiple user types for example administrators of charities, so charities can add their own content.

A User manual

When the user opens the app for the first time, they can either:

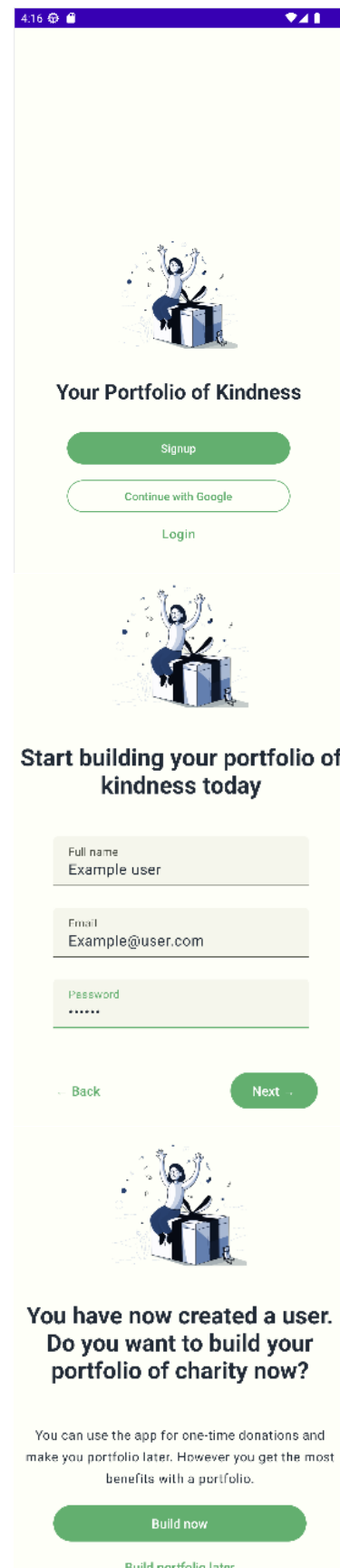
- Sign in.
- Sign in with Google.
- Register.

If the user chooses to sign up, they are first asked to enter their personal information. The name can be anything, but the e-mail must be entered as e-mail regex and the password must be at least 6 characters long. Once the

user has entered valid information and clicked on "Submit", the user will be presented, where they can either:

- Build their portfolio now.
- Build their portfolio later.

which will take the user to the end of the registration process.



The image shows a mobile app registration flow on a yellow background. At the top, a purple status bar shows the time 4:16 and battery level. The first screen is titled "Your Portfolio of Kindness" and features an illustration of a person sitting on a large gift box. Below the illustration are three buttons: a green "Signup" button, a white "Continue with Google" button with a green border, and a green "Login" button. The second screen is titled "Start building your portfolio of kindness today" and features the same illustration. Below the illustration are three input fields: "Full name" with the placeholder text "Example user", "Email" with the placeholder text "Example@user.com", and "Password" with a green border and six dots. Below the input fields are two buttons: a green "Back" button and a green "Next" button. The third screen is titled "You have now created a user. Do you want to build your portfolio of charity now?" and features the same illustration. Below the illustration is a paragraph of text: "You can use the app for one-time donations and make you portfolio later. However you get the most benefits with a portfolio." Below the text are two buttons: a green "Build now" button and a green "Build portfolio later" button.

4:16

Your Portfolio of Kindness

Signup

Continue with Google

Login

Start building your portfolio of kindness today

Full name
Example user

Email
Example@user.com

Password

Back Next

**You have now created a user.
Do you want to build your
portfolio of charity now?**

You can use the app for one-time donations and make you portfolio later. However you get the most benefits with a portfolio.

Build now

Build portfolio later

If the user chooses to build their portfolio now, they will be prompted to select the amount they wish to donate and how frequently they wish to donate.

Next, the user can choose which charities they wish to be subscribed to.

They can do this by selecting the organizations.

The user is then shown a summary of their portfolio, and by clicking on

"Continue", the registration process is completed.

How much do you want to donate

50
100
200

Back Next

How often will you make a donation

Monthly
Quarterly
HalfYearly
Yearly

Back Next

Explore the different charities

All Health Disasters Climate Welfare Chi

Red Barnet

Children in the world must not grow up in insecurity or poverty. We create better lives for poor

Red Cross

founded to protect human life and health, to ensure respect for all human beings, and to prevent and

Knæk Cancer

We support cancer research and help children and adults in hospitals with their stay.

UNICEF

At UNICEF, we want to create better conditions for children everywhere in the world. No child should

WWF Nature

Back Next

Great! You are now set to donate 50 kr. per month to the following charities

Adjust the percentages to each organization below.

Organization	%	DKK
Red Barnet	50%	25 kr.
Red Cross	50%	25 kr.

Pr. month donation: 50 kr

☒ Accept terms of service.

After logging in, the user will be redirected to the home page, where they can see:

- Articles from charities they subscribe to.
- Charities they can follow.

At the bottom of the screen, the user can use the navigation bar to navigate the app. When the user navigates to the portfolio page, they will see:

- Their monthly donations are in a pie chart.
- A table more detailed information.

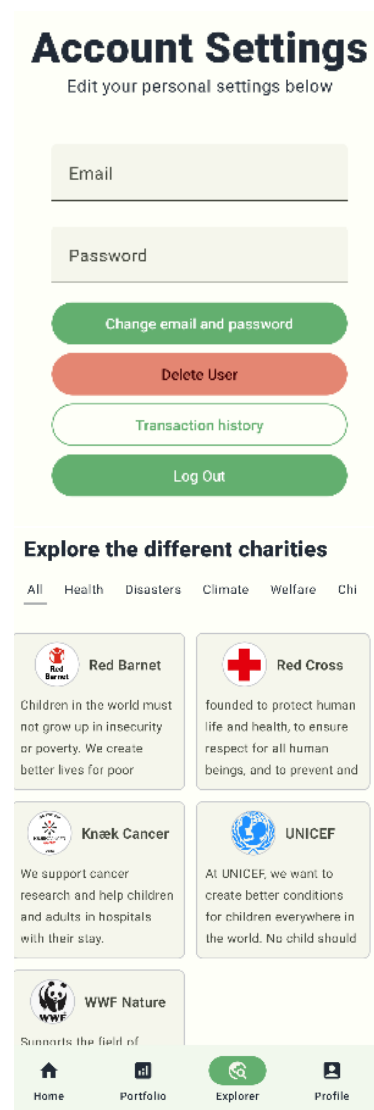
The screen also includes a FAB, which allows the user to edit subscription values.



On the explorer screen, the user can find various charities and search for charities in specific categories using the filter at the top of the screen.

On the profile screen, the user can:

- Edit their account information.
- Delete the account.
- Log out
- View all transactions



B Time Spent

We've included a timetable of the estimated time spent on each user story. It's important to note that the timetables do not accurately depict the work done as some tasks aren't directly related to any user stories, as well as the timetable does not factor in report writing, project planning, etc.

US1: Build portfolio					
	August	Espen	Mads	Mark	Lucas
Analysis	4	4	0	2	2
Design	2	5	5	3	0
UX	1	4	2	1	1
Front-end Implementation	0	0	7	4	5
Back-end Implementation	4	0	10	15	10
Refining/Debugging	3	0	5	3	5
Testing	0	1	2	1	1
Sum	14	15	31	29	24

US2: Edit portfolio					
	August	Espen	Mads	Mark	Lucas
Analysis	4	4	0	2	2
Design	2	2	5	5	0
UX	1	4	2	4	1
Front-end Implementation	2	5	6	12	5
Back-end Implementation	2	1	5	9	4
Refining/Debugging	1	1	1	1	1
Testing	1	1	1	1	1
Sum	13	18	20	21	14

US3: One-time donation					
	August	Espen	Mads	Mark	Lucas
Analysis	0	0	1	0	0
Design	0	0	1	2	0
UX	0	1	3	2	1
Front-end Implementation	0	3	5	0	0
Back-end Implementation	0	0	18	0	0
Refining/Debugging	0	0	4	0	0
Testing	0	1	3	1	1
Sum	0	5	35	5	2

US4: Subscribe to charities					
	August	Espen	Mads	Mark	Lucas
Analysis	4	4	0	0	3
Design	1	1	2	2	2
UX	1	4	2	1	1
Front-end Implementation	0	4	2	2	5
Back-end Implementation	1	0	5	6	11
Refining/Debugging	1	0	1	1	1
Testing	1	1	1	1	1
Sum	9	14	13	13	25

US5: Explore charities					
	August	Espen	Mads	Mark	Lucas
Analysis	2	2	0	0	3
Design	1	4	2.5	3	0
UX	1	1	3	3	2
Front-end Implementation	2	7	7	8	8
Back-end Implementation	4	0	15	9	17
Refining/Debugging	2	1	2	3	7
Testing	4	1	3	1	4
Sum	12	16	31	25	30

US6: Read charity news					
	August	Espen	Mads	Mark	Lucas
Analysis	3	0	2	0	0
Design	0	0	0	4	2
UX	1	1	2	3	4
Front-end Implementation	1	5	5	6	0
Back-end Implementation	4	0	2	16	11
Refining/Debugging	1	0	1	1	1
Testing	1	1	1	1	1
Sum	11	7	13	30	19

US7: Sign up					
	August	Espen	Mads	Mark	Lucas
Analysis	4	2	0	0	3
Design	2	1	2	2	2
UX	2	2	2	1	1
Front-end Implementation	6	6	2	5	4
Back-end Implementation	5	0	5	0	0
Refining/Debugging	3	0	1	1	1
Testing	3	1	1	1	1
Sum	25	12	13	10	12

US8: Login					
	August	Espen	Mads	Mark	Lucas
Analysis	3	1	3	1	1
Design	3	0	3	2	2
UX	2	2	4	1	1
Front-end Implementation	5	6	2	4	3
Back-end Implementation	3	0	3	3	5
Refining/Debugging	0	0	3	1	1
Testing	2	1	2	1	1
Sum	18	10	20	13	13

US9: View billing history					
	August	Espen	Mads	Mark	Lucas
Analysis	0	0	1	1	1
Design	0	0	1	1	1
UX	0	0	2	1	1
Front-end Implementation	2	0	0	0	0
Back-end Implementation	4	0	5	0	0
Refining/Debugging	2	0	3	0	0
Testing	2	1	2	1	1
Sum	10	1	14	4	4

C Composable Function implementation example

The full code can be found in: `app/src/main/java/com/example/kind/view/composable/KindTextField.kt` Kind-TextField class holds the state and the composable function and contains all the logic for validating fields in our app.

```

1 class KindTextField(
2     ...
3 ) {
4     var text: String by mutableStateOf("")
5     var label: String by mutableStateOf(label)
6     var errorText: String by mutableStateOf("")
7     var hasError: Boolean by mutableStateOf(false)
8
9     @Composable
10    fun Content() {
11        TextField(
12            value = text,
13            isError = hasError,
14            label = { Text(text = label) },
15            modifier = Modifier.padding(10.dp),
16            onValueChange = { value ->
17                hideError()
18                text = value
19            },
20            singleLine = true,
21            supportingText = { if (hasError) Text(text = errorText) },
22            ...
23        )
24    }
25
26    fun validate() {...}
27    ...
28 }

```

D Viewmodel implementation example

The full code can be found in: `app/src/main/java/com/example/kind/viewModel/ArticleViewModel.kt`

```

1 class ArticleViewModel(
2     ...,
3     val storage: StorageServiceImpl,
4     val id: String,
5 ) : ViewModel() {
6
7     // State setup
8     private val _data = MutableStateFlow(Article())
9     val data: StateFlow<Article> = _data.asStateFlow()
10
11    // Fetch data by instantiation
12    init {
13        getArticleById()
14    }
15
16    fun getArticleById() {
17        viewModelScope.launch {
18            val articleData = storage.getArticle(id)
19            _data.update { articleData }
20        }
21    }
22 }

```

E Payment intent object

The Payment intent object from Stripe with essential payment information

```

1 {
2   "id": "pi_3MQScyEdqjLxqDOf0M7h0DTm",
3   "object": "payment_intent",
4   "amount": 260,
5   "amount_capturable": 0,
6   "amount_details": {
7     "tip": {}
8   },
9   "amount_received": 0,
10  "application": null,
11  "application_fee_amount": null,
12  "automatic_payment_methods": null,
13  "canceled_at": null,
14  "cancellation_reason": null,
15  "capture_method": "automatic",
16  "client_secret": "pi_3MQScyEdqjLxqDOf0M7h0DTm_secret_vsFPwSge33uqCnUV0ro32pukx",
17  "confirmation_method": "automatic",
18  "created": 1673775348,
19  "currency": "dkk",
20  "customer": "cus_NATNzQbvJN2rh8",
21  "description": null,
22  "invoice": null,
23  "last_payment_error": null,
24  "latest_charge": null,
25  "livemode": false,
26  "metadata": {},
27  "next_action": null,
28  "on_behalf_of": null,
29  "payment_method": null,
30  "payment_method_options": {
31    "card": {
32      "installments": null,
33      "mandate_options": null,
34      "network": null,
35      "request_three_d_secure": "automatic"
36    }
37  },
38  "payment_method_types": [
39    "card"
40  ],
41  "processing": null,
42  "receipt_email": null,
43  "review": null,
44  "setup_future_usage": null,
45  "shipping": null,
46  "statement_descriptor": null,
47  "statement_descriptor_suffix": null,
48  "status": "requires_payment_method",
49  "transfer_data": null,
50  "transfer_group": null
51 }
```

F Formal Tests

Test id	01.
Requirment	M01.
Scenario	Make one-time donation
Given	I am a user and I want to support a specific charity.
When	I press the 'Donate' button
Then	The system prompts me to enter an amount and confirm the transaction.
Test steps	<ol style="list-style-type: none"> 1. Be on a desired charity page. 2. Press the 'Donate' button. 3. Select an amount. 4. Select a payment card. 5. Press the 'Pay' button
Status	Approved.
Test by	Mark.
Test Environment	Android Studio Electric Eel 2022.1.1.

Test id	02.
Requirement	M06.
Scenario	Add or remove charity from portfolio.
Given	I am a user and remove and add charities from my portfolio.
When	I press the 'Add' or 'Remove' button.
Then	The system adds or removes the charity to my portfolio.
Test steps	<ol style="list-style-type: none"> 1. Be on a desired charity page. 2. Press the 'Add' button to add the charity to my portfolio. 3. Press the 'Remove' button to remove the charity from my portfolio.
Status	Approved.
Test by	Mark.
Test Environment	Android Studio Electric Eel 2022.1.1.

Test id	03.
Requirement	S12.
Scenario	Edit portfolio.
Given	I am a user on the portfolio page. I want to change my donation amount for each charity in my portfolio.
When	I change the donation amounts in a writable field and press a 'Save Changes' button.
Then	The system prompts me to save the changes and verify. Afterwards the changes should be illustrated in my portfolio.
Test steps	<ol style="list-style-type: none"> 1. Be on the portfolio page. 2. Press the 'Edit' button in the bottom corner. 3. Select a charity to edit the amount I donate. 4. Type in the new amount. 5. Press the 'Save Changes' button.
Status	Approved.
Test by	Mark.
Test Environment	Android Studio Electric Eel 2022.1.1.

Test id	04.
Requirement	M08.
Scenario	Login.
Given	I am a user and I want to login on the Kind App.
When	I press the 'Login' button.
Then	The system logs me in with my all my information saved.
Test steps	<ol style="list-style-type: none"> 1. Be on the Login screen. 2. Enter my details. 3. Press the 'Login' button.
Status	Approved.
Test by	Mark.
Test Environment	Android Studio Electric Eel 2022.1.1.

Test id	05.
Requirement	M02 and M03.
Scenario	Read charity updates.
Given	I am a user with a profile subscribed to an organisation.
When	I am on the home page
Then	The system shows me a feed of charity news from organisations in my portfolio.
Test steps	<ol style="list-style-type: none"> 1. Be on the home page. 2. Check if the charity news I receive are from charities I have in my portfolio.
Status	Approved.
Test by	Mark.
Test Environment	Android Studio Electric Eel 2022.1.1.

Test id	06.
Requirement	S14.
Scenario	Update email and password.
Given	I am a user with a profile on my own profile page.
When	I press the 'Update changes' button.
Then	I am given access to edit my email and password, and hereafter save the new wanted changes.
Test steps	<ol style="list-style-type: none"> 1. Be on the profile page. 2. Press the 'Update Changes' button. 3. Fill out new informations. 4. Press the 'Save Changes' button.
Status	Approved.
Test by	Mark.
Test Environment	Android Studio Electric Eel 2022.1.1.

Test id	07.
Requirement	C20.
Scenario	Delete User.
Given	I am a user and I want to delete my account.
When	I press the 'Delete User' button.
Then	I am taken to a verification survey to confirm I want to delete my user and afterwards my user is deleted.
Test steps	<ol style="list-style-type: none">1. Be on the profile page.2. Press the 'Delete User' button.3. Verify my user details.4. Press 'Confirm' button.
Status	Approved.
Test by	Mark.
Test Environment	Android Studio Electric Eel 2022.1.1.

Test id	08.
Requirement	M04.
Scenario	View of donations.
Given	I am a user and I want to see how my donations are distributed.
When	I am on the portfolio page.
Then	I see illustrated and organized data with my donations.
Test steps	<ol style="list-style-type: none">1. Be on the portfolio page.2. View the data.
Status	Approved.
Test by	Mark.
Test Environment	Android Studio Electric Eel 2022.1.1.

Test id	09.
Requirement	M05.
Scenario	Create a profile.
Given	I am a user and I want to create an account in the Kind app.
When	I am on start screen.
Then	I am taken to a sign up part, where I can create my account.
Test steps	<ol style="list-style-type: none">1. Be on start screen.2. Press the 'Sign Up' button.3. Go through the sign up flow and enter my desired information.
Status	Approved.
Test by	Mark.
Test Environment	Android Studio Electric Eel 2022.1.1.

Test id	10.
Requirement	M07.
Scenario	Reset password.
Given	I am a user and I want reset my password.
When	I am on the login screen and press forgot password.
Then	I am taken to screen where I have to enter my Email so I can reset the password.
Test steps	<ol style="list-style-type: none">1. Be on login screen.2. Press the 'Forgot Password' button.3. Enter my Email.4. Reset password in my Email.
Status	Approved.
Test by	Mark.
Test Environment	Android Studio Electric Eel 2022.1.1.