March 31, 2021

PAY UP

FYP REPORT

EOIN GOHERY
17206413
LM110 - Computer Games Development

Pay Up

A New Approach to Facilitate Recurring Payments

A Final Year Project Presented

by

Eoin Gohery

17206413

In fulfilment of the requirements for the Degree of Bachelor of Science in Computer Games Development

Supervisor: Chris Exton

Date: 31st March 2021

TABLE OF CONTENTS

TABLE OF TABLES

TABLE OF FIGURES

ABSTRACT

In this paper, I present the research, design, and evaluation of an android application, that provides an alternate and unified process to facilitate the splitting of recurring expenses within a group. Subscription plans such as Netflix Premium, Spotify Family or any other multi-user subscription-based service have risen as an industry-standard business model for providing a service to a group of individuals under a singular account.

In many instances, the payment for the account is taken from a single account. This leaves the primary account owner of such a service to decide the amounts and methods in which the beneficiaries of the subscription are to pay their share. This project is an attempt on my part to allow for p2p payments directly to the bank account of the subscription owner from multiple payment sources including Direct Debit, Bank Transfer, and especially mobile wallets

ACKNOWLEDGEMENT

It is with immense gratitude that I acknowledge the support and help of my Supervisor, Dr Chris Exton, who has always encouraged me throughout research and development. Without his continuous guidance and persistent help, this project would not have been a success for me. I would also like to thank my friends and classmates for all the help and support they have given me during this project and over the past four years. I am grateful to the University of Limerick and the department of Computer Science without which this project would have not been an achievement.

AUTHOR'S DECLARATION

This final year project is presented in part fulfilment of the requirements for the degree of Bachelor of Science in Computer Games Development. It is entirely my own work and has not been submitted to any other University or higher education institution, or for any other academic award in this University. Where there has been use made of the work of other people it has been fully acknowledged and fully referenced. All brands, product names, or trademarks are properties of their respective owners.

A license for the application icon was purchased from clipdealer.com. With this purchase, I have full rights to reuse this image for personal or commercial use, with and without modification. This license can be found in the docs folder of GitHub.

SIGNED: 

DATE: _____

LIST OF ABBREVIATIONS

p2p	Peer-to-Peer
OS	Operating System
iOS	iPhone Operating System
API	Application Programming Interface
app	Application
admin	Administrator
FYP	Final Year Project
P2PM-pay	Peer-to-Peer Mobile Payment System
mP2P	Mobile Peer-To-Peer
e-wallet	Electronic Wallet
FDD	Feature-Driven Development
JWTs	JSON Web Tokens
GUI	Graphical User Interface
APK	Android Package Kit
AAP	Android Application Package
SEPA	Single Euro Payments Area
EPS	Electronic Payment Standard
URL	Uniform Resource Locator
Uri	Universal Resource Identifier
CLI	Command Line Interface/Interpreter
uid	unique identifier
id	unique identifier
APK	Android Package Kit
HTTP	Hypertext Transfer Protocol

Table 1: List of Abbreviations

Report

1 INTRODUCTION

1.1 RESEARCH QUESTION

How could a peer-to-peer payment system that consolidates payment methods, assist in the managing of recurring group expenses?

1.2 CONCEPT

The application consists of three major components: 1) a payment system combined with a secure payment checkout for allowing p2p payments, 2) an authentication system, and 3) an easy-to-use interface.

I hope to create a singular environment in which the beneficiary of the service can pay the primary account owner in whichever manner the beneficiary sees fit while allowing the account holder to receive the payment directly into their bank account automatically.

“The peer-to-peer mobile payment (P2PM-pay) system is a new form of technology expected to grow dramatically in the market in the next few years.” (Zoran Kalinic, 2019)

1.3 MOTIVATIONS

1.3.1 Digital Payments

I am interested in exploring the methods in which digital payments are implemented into software and the security features that are applied to allow for safe a secure transfer of funds. The exchange of money itself has progressed significantly in the previous years towards a cashless society. We are beginning to see an almost abstract conception of money among the general population. With that in mind, I see this project as an excellent opportunity to explore digital payments as a growing trend and implement my variation on this concept.

1.3.2 Payment Gateway for P2P

Many payment gateways have been developed that allow for direct p2p payments. But many of these gateways only allow for the payment to be transferred between two accounts of that service. PayPal is an example of this as although they allow payments to be made securely through their service, the sender must also possess a PayPal account

and the money that is transferred is stored within the PayPal service, to be withdrawn later.

Other such services as stripe do allow for the sender to use their method of payment but are more aimed towards the online sale of services and products. The stripe connect platform allows for p2p payments with the intention that it is used on a marketplace.

I wish to explore the possibility that such a service as Stripe Connect can be used to facilitate p2p payments, not for the sale of a service or product, but as a "middleman" system to bridge the senders preferred payment method and the receivers bank account. All without the receiver requiring the payment method of the sender.

1.4 OBJECTIVES

The primary objectives for this project are as follows:

1.4.1 Research Current Digital Payment Methods

Before any development research will need to be conducted into current payment methods. This includes but is not limited to existing mobile wallets, payment gateways and online markets which deploy a p2p payment strategy.

Also, to be researched is the complexities of finance within software applications. Primarily of which is security, which must be a top priority for any s system which manages the money of a customer and holds no ownership of the customer's money. This is especially important as this prototype will be developed to be a zero-cost service. (No cost for customer or myself as admin)

There is future potential for the application to pull in revenue, but monetary gain is not the focus of an academic project, and as such is not within the scope of this project. I will explore the financial potential of the product upon completion of the prototype (See Further Development)

1.4.2 Design a System

Design a system that utilizes the methods explored in Motivations. This system will be designed for implementation in Android Studio for the Android OS. The design of the system must consider the requirements, architecture, and methods to be used for implementation. The APIs used for this system are determined in the research stage. Each aspect and feature of this application must be thoroughly researched and designed before implementation. This is to prevent and fundamental flaws in the methods that are used.

1.4.3 Implement a Prototype

Once the design of the system is complete, the next objective is to implement an intuitive p2p payment system that incorporates multiple payment methods as per the methods to be researched. This implementation of a prototype phase will be heavily guided by the design created. Any changes that are made in this phase will be retroactively fitted into the design documentation. This is to ensure that any minor scope changes are represented throughout the final product report.

1.5 CONSISTENCY IN BOTH DESIGN AND IMPLEMENTATION WILL BECOME VITAL FOR THE TESTING AND EVALUATION OF THIS PROJECT. METHODS FOR TESTING WILL BE DECIDED UPON IN THE TESTING PHASE OF EACH FEATURE. (SEE RECORD PAYMENT OUTSIDE APP

For the recording of payments received outside of the app, I simply added a received button to the payment object displayed on the incoming screen. The button when pressed updates the incoming document of that payment. It updates the fields; active: true, “payment_method”: Outside of App, and “date_paid” to the current system DateTime. These changes trigger the MarkasRecieved cloud function which then updates the respective due document of that payment.

1.6 SECURITY

1.6.1 Database Rules

Figure 43: Security Rules

The rules for the Firestore Database are quite simple. They allow for any user to read from the database but only allow the users to write to the database of their document. When a payment is created in the incoming subcollection, this triggers a cloud function that creates the corresponding document in the beneficiaries document. The cloud functions on the server are run with administrator rights and can affect all documents freely.

1.6.2 Strong Customer Authentication (SCA)

“Starting September 14, 2019, new payments regulation is being rolled out in Europe, which mandates Strong Customer Authentication (SCA) for many online payments in the European Economic Area (EEA).” SCA requires two-factor authentication on payments.

There are exceptions to this in which the additional authentication n is not required. IN most cases these exceptions are payments deemed as low risk. For Stripe Low risk is determined by a built-in risk analysis tool which among other things considers the amount to be paid. Most payment under the value of 30 EUR will be regarded as except unless otherwise flagged as risky by the bank. In this case, authentication will then be required. Which Stripe provides in the form of “3D Secure 2”.

1.6.3 3D Secure 2

This form of authentication is designed to provide a “frictionless authentication” method. In the instance of an android application, it is represented as a built-in authentication activity. For the testing mode, this application appears as Figure 44: 3D Secure 2 on the right. If the app were to go live, this would appear as a message indicating for the user to go to the banking application for the bank in which they are using. It may also request a one-time passcode which will be sent to the user's registered mobile number from their chosen bank.

“Other card-based payment methods such as Apple Pay or Google Pay already support payment flows with a built-in layer of authentication (biometric or password).” In testing Google Pay requires a biometric fingerprint input or password input in the payment overlay that appears when selecting Google Pay.

Testing)

1.7 DEVELOPMENT METHODOLOGY

Feature-Driven Development, an iterative and incremental approach to software development, is derived from the Agile methodology. (Nawaz, 2017) There are five steps to FDD.

1. Develop an overall model.
2. Build a feature list.
3. Plan by feature
4. Design by feature.
5. Build by feature.

Additional: Test and Review (Not traditionally a step in FDD but one I shall be using)

Steps 4 and 5 will take up most of the effort. After each feature, status reporting is conducted to help track progress, results, and possible errors. I believe this method will be appropriate for a single person development team as it will allow me to focus all my attention on a particular feature. Review and test said feature and then moving on to the next feature. This will hopefully ensure consistent quality throughout all features.

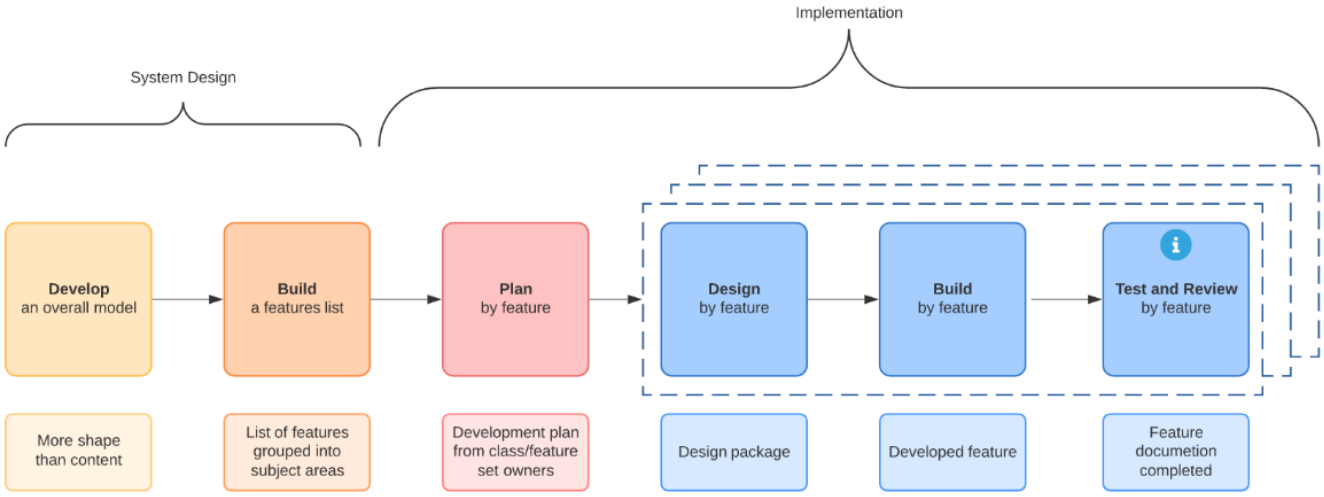

Figure 1: FDD Model

1.8 PROJECT PLAN

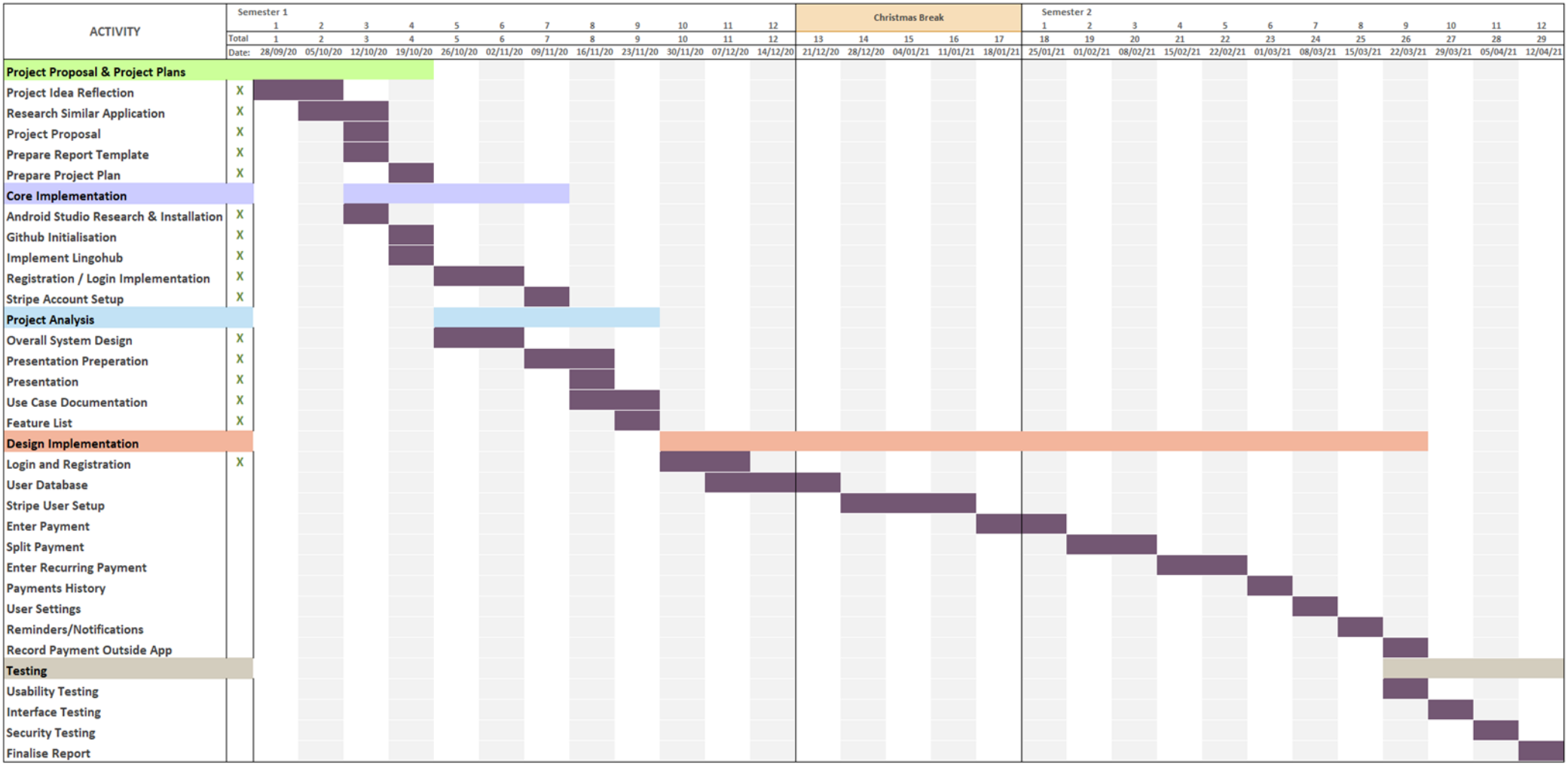

Figure 2: Project Plan

2 BACKGROUND

2.1 SUBSCRIPTION SHARING CULTURE

With the rise of subscription-based payment models, applications have evolved to make use of this method as a primary source for generating revenue.

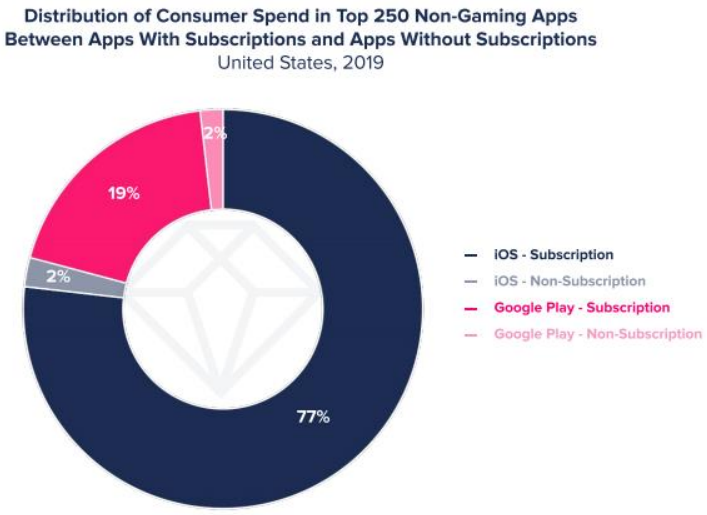

Figure 3: Subscription Services within Applications (App Annie, 2020)

But how popular have these subscription services become? Figure 3 to the right details the results of one study conducted by App Annie. This study reports "In-App Subscriptions Contribute to 96% of spending in Top Non-Gaming Apps" (App Annie, 2020)

This further highlights the growing trend of applying subscription fees to applications as a primary method for the sale of the service.

But so many subscription services came the trend of sharing accounts. Most of the larger subscriptions such as Netflix and Spotify have ratified this issue by created group-based plans. While this allows for multiple users (usually restricted to family or same residence) to access the service without breaching terms of service, these companies have not implemented a method for the multiple individuals to pay for the service. This falls to one single "account holder".

A study conducted by Survata involving 2,255 people in the US, found that 31% of Americans who have access to a Netflix subscription are utilising the service without paying for it. (Epstein, 2016).

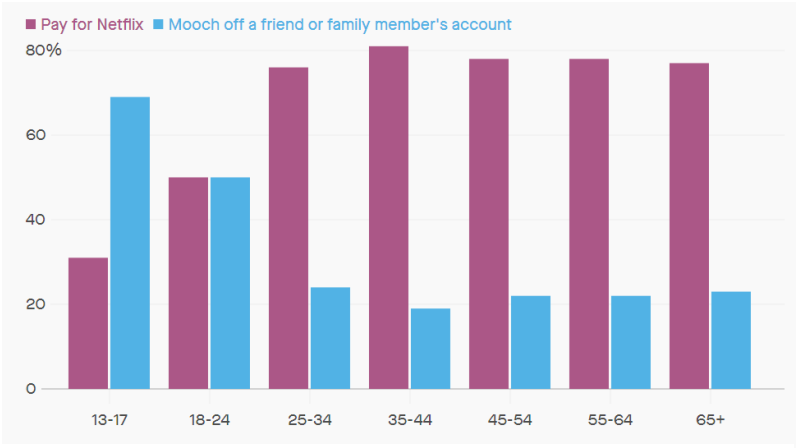

Figure 4: Proportion of users who pay for Netflix (Epstein, 2016)

Other studies reported that “About 66% of Gen Z and 64% of millennials reported using someone else’s subscription service, compared to just 34% of Gen Xers and 26% of baby boomers.” (Moeser, 2020) This is expected as many of the studies in this report have indicated a greater level of acceptance for digital services in Gen Z’s and Millennials.

It has also been found that “the most common people they borrow passwords from our friends (29%), significant others (29%), parents (28%) and siblings (26%).” (Moeser, 2020). As previously stated most subscription services only offer these multi-user accounts on the requirement that the users are related or living in the same residence. Yet there are no methods implemented in which this is enforced, leading to a greater amount of sharing, as indicated by a 29% borrow rate for friends above. Figure 5 below indicates that of a recent survey of multiple age categories that 42% have borrowed password to access a subscription service with a high majority in video streaming services.

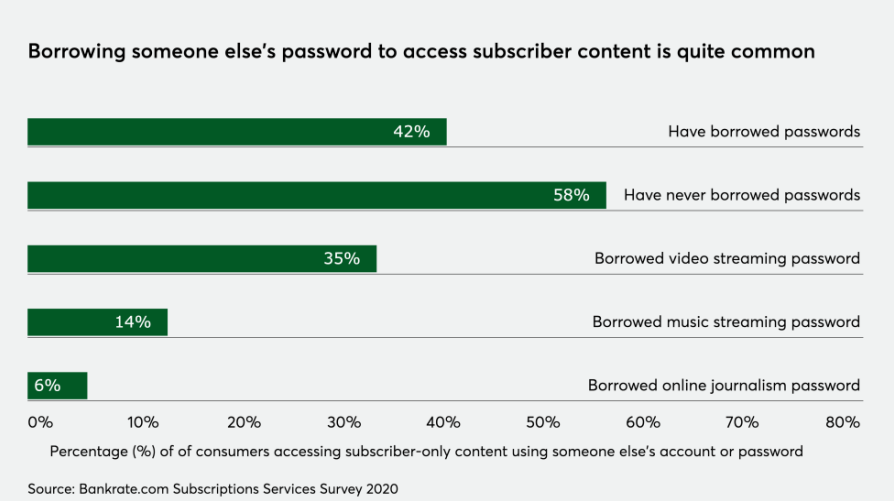

Figure 5: Users borrowing passwords

In particular, Video and Music streaming services have skyrocketed in popularity.

Service Name	Max Users	Concurrent Streams
Netflix	4	4
Disney+	7	4
Amazon Prime Video	6	3
Now TV	6	2
HBO NOW	N/A	3
Apple TV+	6	6
Spotify	6	6
YouTube Music	6	6

Table 2: Subscription service user limits

More services are expected to launch in the coming years such as HBO MAX. All of these services offer concurrent stream subscriptions that must be paid out from a singular payment source. This leaves the account admin with the task of collecting the required payments from potentially 6 other individuals. This leads to the question, what is the best way to collect the money owed? The likelihood that all of these individuals will pay you on time, in a secure manner with the same payment method is extremely rare.

2.2 RISE OF DIGITAL PAYMENTS AND FALL OF CASH PAYMENTS

When considering digital payment applications and their usage today, one must look at the current trends in banking as it stands today. “Globally, Consumers accessed Finance apps over 1 trillion times in 2019, up 10% from 2017. From stock management to mobile banking and payment apps, this showcases mobile’s central role in managing our daily finances” (App Annie, 2020). With a greater increase in mobile usage and minor advancement in online banking, new methods for managing money online are receiving a much higher level of recognition than previously before.

Before Covid-19, the percentage of payments that were cash payments in the UK was 58% in 2009, 48% in 2014 and 23% in 2019. (Tony Chen, 2018) This continuously diminishing trend will only have been amplified further in light of recent events. In the wake of Covid-19, cash in hand has received a critical blow. Even as the pandemic reaches a social end (distinct from a medical end), the heightened awareness of cleanliness remains with the general public. The use of cash will be set to fall even further in the coming years, opening for a greater rise in P2PM-pay adoption.

One report which consolidated data of 16 separate studies from 2015 to 2018 used artificially trained neural networks to determine trends in acceptance of p2p mobile payments. Upon review of their findings, they discovered that “perceived usefulness still has the strongest influence on intention to use P2PM-pay” (Zoran Kalinic, 2019), This study suggests that for any p2p application to be successful, the usefulness of the application must be a top priority while perceived safety followed as the second most important factor. For some studies, security and trust were found to be of the highest priorities (Shin, 2009) or second only to price. (Edelman Intelligence, 2020) For perceived safety/security to be demonstrated it will be vital to indicate the method by which the transfer takes place. Trust in the name of a well-known brand such as a large bank or payment gateway at checkout will serve to provide this perceived safety. Because of this, extra care must be taken in the selection of the chosen payment gateway.

2.3 TECHNOLOGIES

The technologies that I will implement the features within this application are as follows:

2.3.1 Android

According to the eMarketer study released in 2020, adult smartphone users spend about 4 hours each day on mobile internet, and 88% of the time on mobile applications instead of browsers. (Wurmser, 2020) This highlights the significance of utilising a smartphone application rather than a website, and therefore the need to ensure innovative, responsive applications.

In terms of this project, Android applies to the smartphone operating system developed by Google and based on the Linux kernel. Its primary application is for handheld touchscreen devices such as smartphones and tablets, but it can also be used to produce special user interfaces for televisions and digital watches. In June 2020, the Google Play store reached 2.96 million Android applications launched, with a peak of over 3.5 million in 2017. More than 100,000 Android applications are released on the Play Store each month. (MindSea Team, 2020)

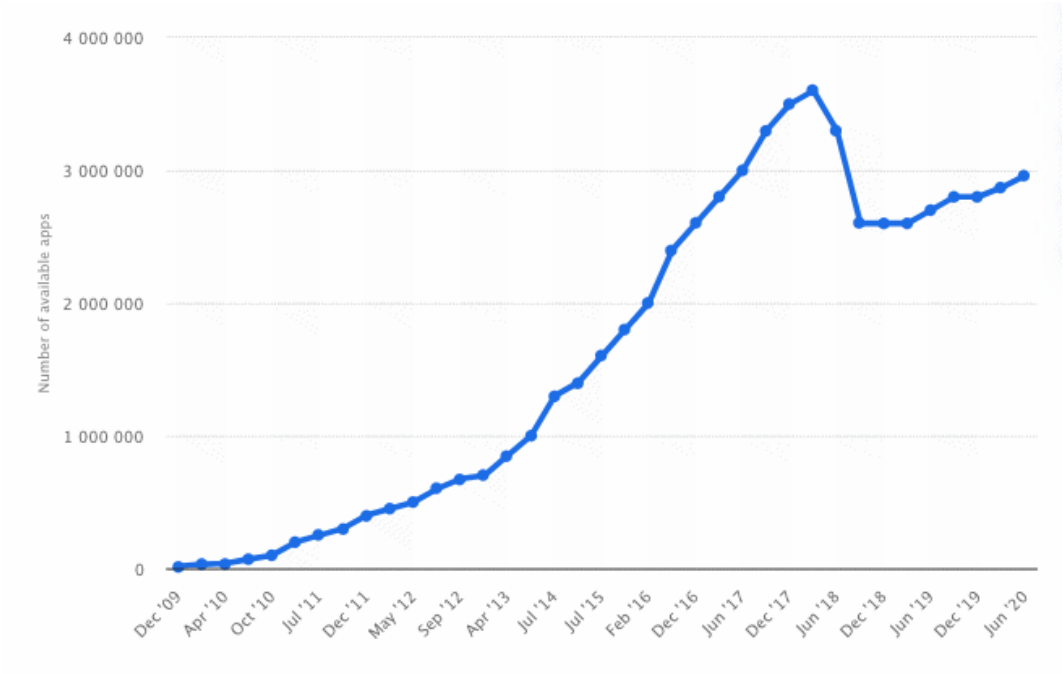

Figure 6: Android Application Quantity (MindSea Team, 2020)

In 2019, Google revealed that there were over one 2.5 billion active monthly Android devices. Google published Android source code under open source licenses, which ensures that it's free for anyone to use it. This culminated in a wider group of developers using open-source code as a base for community-driven projects. Android's popularity has contributed to smartphone wars between technology firms, Google, and Apple.

Despite the success of Apple's iPhone range, Android remained the more user-friendly platform for application development with Apple's iOS App Store focusing more on profits and has monetization opportunities for developers and is more limited in available API's.

The development for this application will be restricted to the google play store only, with an iOS version outside of the scope of this project. As this app is a proof of concept, there is little need for two variations of the app to be developed.

2.3.2 Payment Gateways

When choosing suitable a payment gateway, two options presented themselves as potential services, PayPal, and Stripe. For this selection the reliability should be considered important than the speed of settlement, (Salmony, 2017) Apart from reliability, important factors are; security, ease of use, extendibility, developer documentation and support for multiple payment methods.

2.3.2.1 PayPal

Some studies indicate that a primary issue with mP2P payments is currency conversion (Salmony, 2017) with a particular focus on the currency and geographies of connecting Europe's mP2P systems to the USA's Venmo. PayPal has bridged this gap with its acquisition of Venmo. The primary problem that arises from using PayPal as my payment gateway is that PayPal only accepts payments within its network i.e. both sender and receiver must have a PayPal account. This is not within line with the scope of this project and will not be suitable for this application. That is not to say that PayPal could not be explored as a potential feature. Multiple payment gateways can be used to extend potential payment methods. (See Further Development PayPal)

2.3.2.2 Stripe

Stripe advertises itself as "A fully integrated suite of payments products" on their homepage and from deeper review, this seems to be an accurate statement. Stripe is a globally recognised and trusted payment gateway with high levels of security. (See Security) The Stripe Connect Platform of Stripe in particular will be the focus for my application. Stripe Connect was developed for online marketplace platforms and will serve as an excellent zero-cost solution for a P2P payment system. The standards accounts will serve the purposes of this app without any additional fees. Although the custom and express accounts provide extensive customisation and faster setup, it is not necessary for the prototype. The additional features will be reviewed upon development completion. (See Further Development)

	DESCRIPTION	SUPPORTS RECURRING PAYMENTS	SUPPORTS REFUNDS	SUPPORTS DISPUTES	PAYMENT CONFIRMATION
CARDS	Cards are linked to a debit or credit account at a bank. To complete a payment online, customers enter their card information at checkout.	Yes	Yes	Yes, highest dispute rate	Immediate
WALLETS	Wallets are linked to a card or bank account, but can also store monetary value. Wallets typically require customer verification (e.g., biometrics, SMS, passcode) to complete a payment.	Yes	Yes	Yes, lower dispute rate than cards	Immediate
BANK DEBITS	Bank debits pull funds directly from your customer's bank account. Customers provide their bank account information and typically agree to a mandate for you to debit their account.	Yes	Yes	Yes, lowest dispute rate	Delayed
BANK REDIRECTS	Bank redirects add a layer of verification to complete a bank debit payment. Instead of entering their bank account information, customers are redirected to provide their online banking credentials to authorize the payment.	No, but Stripe supports recurring for some methods by converting to direct debit	Yes	No	Immediate
BANK CREDIT TRANSFERS	Credit transfers allow customers to push funds from their bank account to yours. You provide customers with the bank account information they should send funds to.	No	Yes	No	Delayed
BUY NOW, PAY LATER	Buy now, pay later is a growing category of payment methods that offers customers immediate financing for online payments, typically repaid in fixed installments over time.	No	Yes	Yes, most methods will take on fraud risk	Delayed
CASH-BASED VOUCHERS	With cash-based vouchers, customers receive a scannable voucher with a transaction reference number that they can then bring to an ATM, bank, convenience store, or supermarket to complete the payment in cash.	No	No	No	Delayed

Table 3: Stripe Payment Methods (Stripe, 2020)

Stripe also allows for a wide variety of payment methods (See Table 3: Stripe Payment Methods . For all the factors mentioned, Stripe Connect is the payment gateway that I will use to facilitate payments.

2.3.3 Mobile Wallets

Within the UK, “Nearly ten million people, or 18% of the adult population registered for mobile payments in 2019 with 79% of these registered users recording a payment. Nearly half (48%) of registered users made payments fortnightly or more frequently” (UK Finance, 2020). This same study found that the likelihood for a younger individual to use one of the mobile wallets researched below was far higher than older individuals. Brand identity for these mobile wallets is key in their uptake as “the acceptance of the mobile wallet is influenced by trust.” (Shaw, 2014). Trust is also increased significantly by the addition of biometric locks (face, fingerprint, or iris scan) for mobile wallets. These mobile wallets currently stand as the three, leading applications in the mobile wallet market.

2.3.3.1 Apple Pay

"Apple Pay is only beneficial for Apple device consumers," says Jared Weitz, CEO, and founder of small-business lender United Capital Source. I am not be developing the prototype for apple devices nor am I in possession of such a device. For this reason, I will not be implementing it into the application prototype.

2.3.3.2 Samsung Pay

Samsung pay has yet to be released for Ireland and any attempts to use the app are blocked. For this reason alone, I will not be implementing Samsung pay into my app prototype as I will have no method to test the success of the integration.

2.3.3.3 Google Pay

Google pay is the only mobile wallet that I will implement into the prototype as it is the only one that I can effectively use and test myself. The implementation for Apple Pay and Samsung Pay is a simple configuration in Stripe Connect that can be easily be implemented at a later date, in a similar method to Google Pay. Although, without the ability to effectively test it, I do not believe it would be of any benefit to the prototype of this “proof of concept” project.

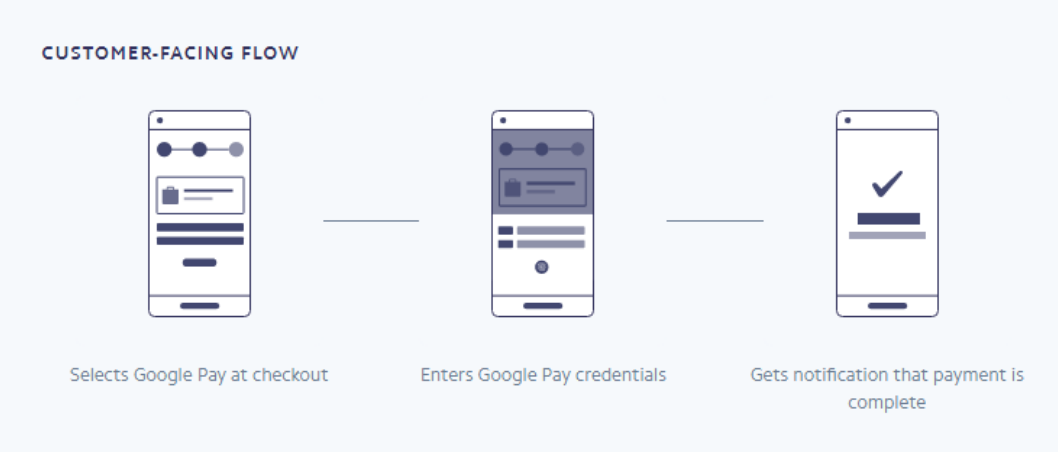

Figure 7: Google Pay Checkout (Stripe, 2020)

2.3.4 Other Payment Methods

2.3.4.1 Card

Debit card transactions remain the most popular method of payment with 4.5 billion payments made in 2019 in the UK alone (UK Finance, 2020). The prototype will allow for Visa, Mastercard and American express, globally with automatic currency conversion.

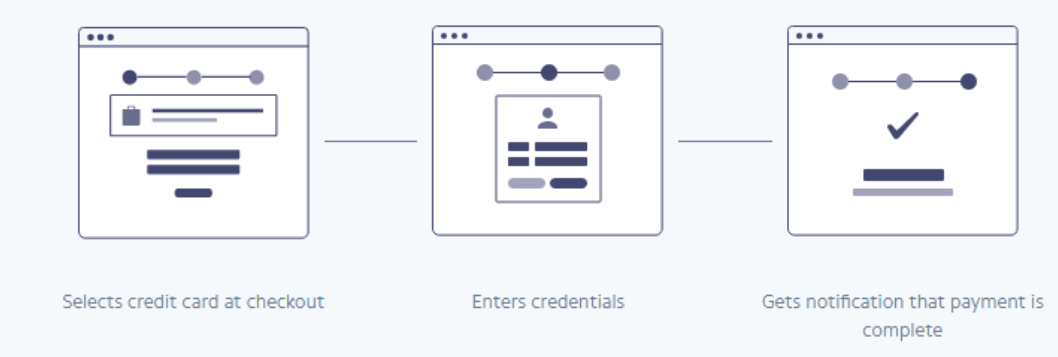

Figure 8: Card Checkout (Stripe, 2020)

2.3.4.2 Direct Debit

SEPA Direct Debit Allows for customers in the Single Euro Payments Area to pay directly from a bank account within European Union. Other direct debit systems will also be available upon checkouts such as iDEAL giropay, EPS and sofort. These combined should allow for direct debit payments throughout all of Europe.

2.4 SECURITY

Safety standards have two key components: physical security for tokens shared between the phones; and user security for the series of procedures that the user would perform to execute the transaction. Both are important elements for a safe payment mechanism. A highly secure token is worthless if the user protocol enables users to pay unwanted people

and a decent user protocol is undermined if the tokens and/or communication mechanisms can be easily tampered with.

2.4.1 Payment Token Security

To protect against credit card identity leakage, leading e-wallets use tokenization. Tokenization safely stores personal payment data in a specially created and single-purpose secure token vault. The details tokenized in digital wallets offer everything required for safe transfers while leaving fraudulent fraudsters with none of the worth. (Worldpay Editorial Team, 2019)

Stripe's method for tokenization is as follows:

- Credentials are sent to Stripe.
- Stripe tokenizes the data and returns a token to the back-end.
- The back-end creates a charge.
- The data is sent to Stripe again, and it shares the details with payment systems.
- Payment systems respond to Stripe and state whether everything alright. Or report about issues.
- Stripe responds to the server about the state of the transaction.

"All card numbers are encrypted at rest with AES-256. Decryption keys are stored on separate machines. None of Stripe's internal servers and daemons can obtain plaintext card numbers but can request that cards are sent to a service provider on a static allow list. Stripe's infrastructure for storing, decrypting, and transmitting card numbers runs in a separate hosting environment and doesn't share any credentials with Stripe's primary services" (Stripe, 2020)

Stripe also uses two PGP keys to encrypt communications with Stripe, a general PGP key for contact with Stripe and a data migration PGP key for sensitive data such as card details.

2.4.2 User Security

User Security is managed by Firestore Authentication. All passwords are encrypted so that even I as admin will be unable to view them. Firestore also offers the ability to write custom security rules and fully restrict read and write access to the data if needed. Firestore Authentication is essentially a token generator and allows for "complete control over authentication by allowing you to authenticate users or devices using secure JSON Web Tokens" (Google, n.d.)

3 SYSTEM DESIGN

3.1 REQUIREMENTS

3.1.1 Functional Requirements

3.1.1.1 Navigation:

- Seamless transfers between events or pages must be made by the application.
- For the user, distinct pages must be clearly described.
- Features must be visible.
- For users on each page, accessible options should be visible.

3.1.1.2 Database:

- For new user registration, the user database must be permanently open.
- The user database must be updated upon the registration of each new user.
- The user database must be updated upon the change in user information.
- For recording user issues, the issues database must be permanently open.

3.1.1.3 Profile:

- The user has the option of making a custom username and password.
- A login can be performed with a Google account
- A login can be performed with a Facebook account.
- A login can be performed with a Twitter account.
- A login can be performed with a standard email address.

3.1.1.4 Payments:

- The payments must be secure
- History of all payments must be recorded
- All payments must arrive into the recipients account within two days of the payment being sent.

3.1.2 User Requirements

- A consumer must have WI-FI or mobile data on their Android device enabled.
- A new user will be required to register or use the given login form.
- A current user must be able to login with the login method established on registration.
- Users should be able to access the app from any Android device that uses the Android 2.4 (Gingerbread) operating system or later.
- The user must process some form of payment method i.e. bank account or digital wallet
- A user who wished to receive a payment must create a stripe account from within the app or link a pre-existing stripe account.

3.1.3 Environmental Requirements

- A WI-FI network or mobile data must be available.

3.1.4 Non-Functional Requirements

3.1.4.1 Performance & Response Time

- The general advice on response time is 0.1second.
- This limit causes the user to believe that the application reacts instantly.

3.1.4.2 Availability Requirement

- The app will be built for smart devices that can support apps that are developed using Android Studio, this will make the app accessible to a large market.
- The goal is to make it available to IOS users in the future but will not be implemented within the scope of the initial development.

3.1.4.3 Maintainability Requirement

- Users must be given the ability to report to developers any bugs that cause problems to allow for correction.

3.1.4.4 Extendibility Requirement

- After the app is launched it can still be updated with additional features.
- After the initial update, the software development process will not conclude.

3.2 DESIGN AND ARCHITECTURE

The application will use the Google Firebase infrastructure to implement Firebase functionality and use its protected database for storing user information and managing user data. This method is ideal for a project of this scale to provide safe real-time connectivity. As the Firebase admin for this project, I alone have access to the user credentials. The passwords will be encrypted and unavailable to anyone, including me. The application will also use Stripe to manage the payments. Stripe follows strict SCA protocols to ensure that all user payment details are secure.

3.2.1 Feature List

- Login and Registration
- User Database
- Stripe User Setup
- Enter Payment
- Split Payment
- Enter Recurring Payment
- Expense History
- User Settings
- Reminders/Notifications
- Record Payment Outside App

The following diagram is a depiction of the system architecture, the program itself will be hosted locally on a device running Android OS.

3.2.1.1 Architecture Diagram

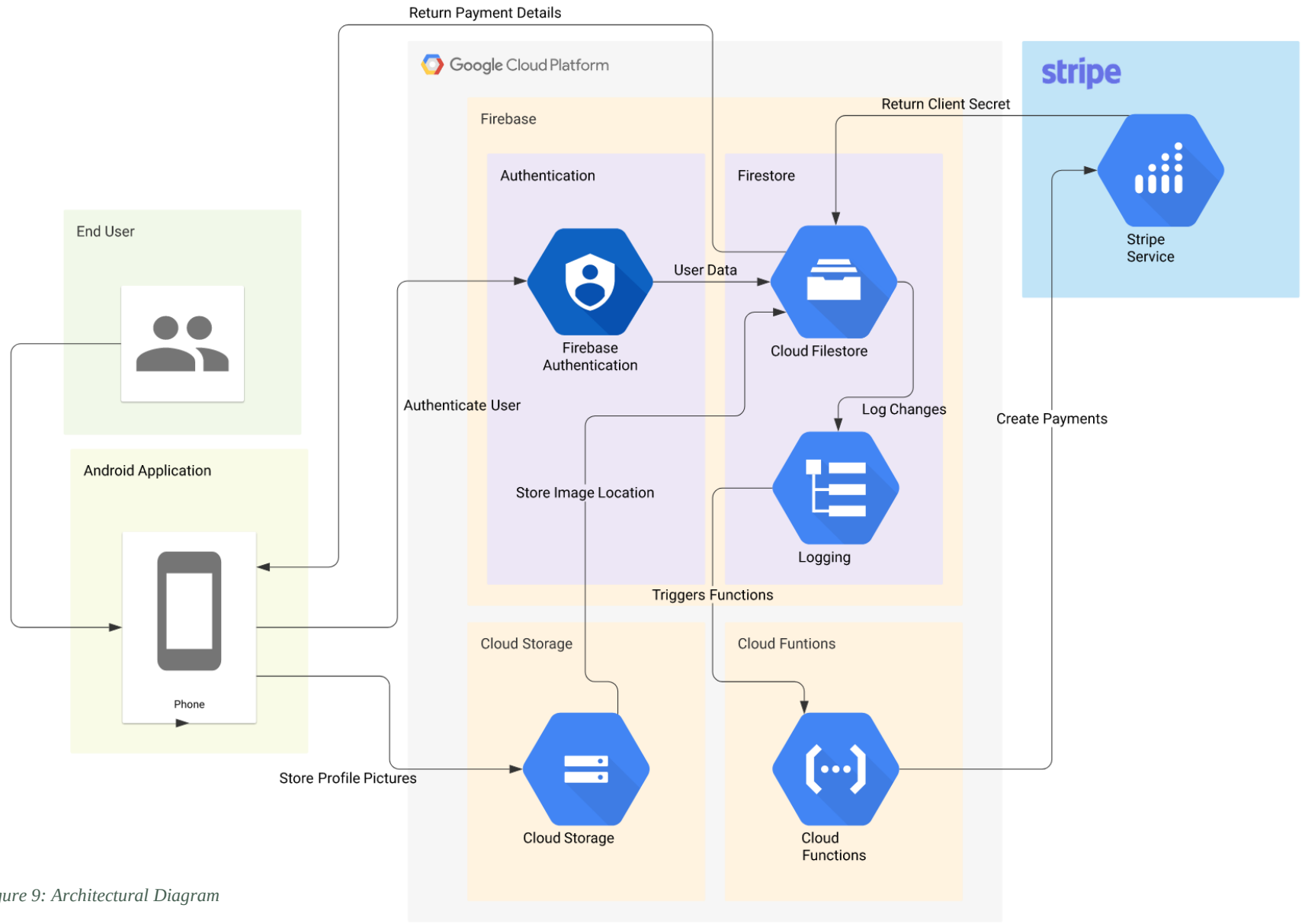

Figure 9: Architectural Diagram

3.3 USE CASES

3.3.1 Use Case 1: Start Application

3.3.1.1 Description & Priority

This use case is first in the priority list and, when the application is started, initiates the core features provided to the user.

3.3.1.2 Scope

The scope of this use case is to explain how the program may be initiated by a user to access the primary functionality available.

3.3.1.3 Use Case Diagram

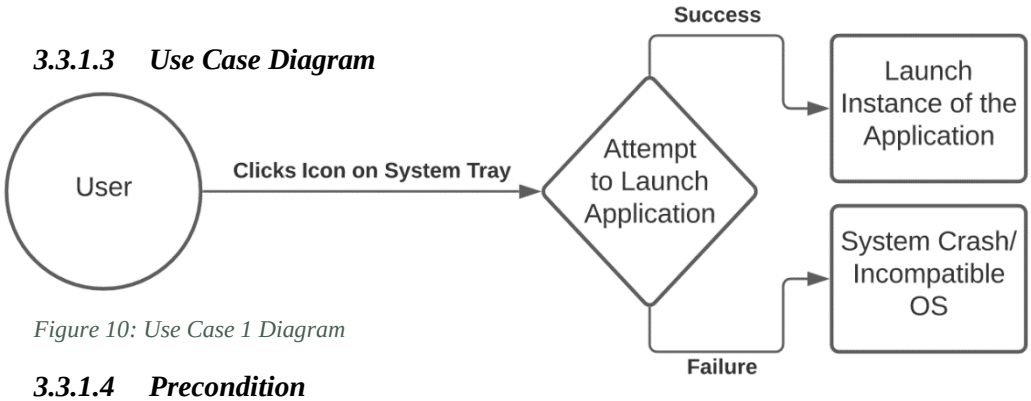

Figure 10: Use Case 1 Diagram

3.3.1.4 Precondition

The user has downloaded and installed the application on a smart device that supports and runs Android OS.

3.3.1.5 Activation

When the user clicks on the icon and launches the application while in its idle state, this usage case is activated.

3.3.1.6 Main Flow

Actor Action: In the Android system menu, a user clicks on the application launcher icon.

System Response: Launch of an instance of the application.

3.3.1.7 Alternate Flow

Actor Action: In the Android system menu, a user clicks on the application launcher icon.

System Response: Due to a system crash or the OS is not compatible, the application does not launch.

3.3.1.8 Terminating

When the application has booted on the device, this flow is terminated.

3.3.1.9 Post Condition

The user is brought to the application's main menu, in which the application enters a state of waiting.

3.3.2 Use Case 2: Register

3.3.2.1 Description & Priority

This use case is second on the priority list as it is required for the core features of the application to be accessible to the user.

3.3.2.2 Scope

The scope of this use case is to access the method in which a new user will create an account within the app.

3.3.2.3 Use Case Diagram

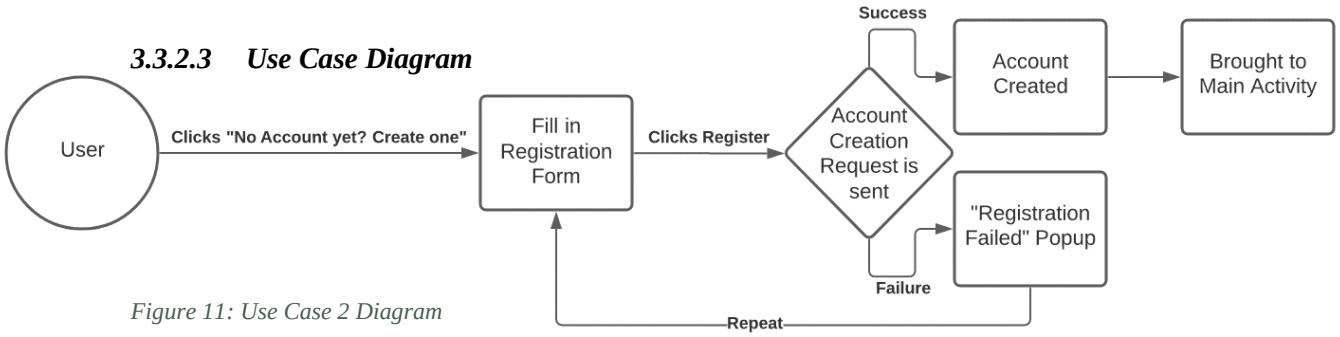

Figure 11: Use Case 2 Diagram

3.3.2.4 Precondition

The user has started the application and has not previously set up an account.

3.3.2.5 Activation

When the user clicks the “No Account yet? Create one.” button text from the login page.

3.3.2.6 Main Flow

Actor Action:

1. User clicks the “No account yet? Create one.” Button.
2. User fills in the required information: email, password, first name, last name.
3. The user clicks the “Register” button.

System Response: Account creation request is sent to the firebase authentication service. Upon success, the user account is created, and the user is logged in.

3.3.2.7 Alternate Flow

Actor Action:

1. User clicks the “No account yet? Create one.” Button.
2. User fills in the required information: email, password, first name, last name.
3. The user clicks the “Register” button.

System Response: Account creation request is sent to the firebase authentication service. Upon failure and “Register Failed” snackbar notification appears.

3.3.2.8 Terminating

When account authentication is successful, this flow is terminated.

3.3.2.9 Post Condition

The user is brought to the main application screen.

3.3.3 Use Case 3: Login

3.3.3.1 Description & Priority

This use case is second on the priority list as it is required for the core features of the application to be accessible to the user.

3.3.3.2 Scope

The scope of this use case is to log in via a user-selected method to access the core features of the app

3.3.3.3 Use Case Diagram

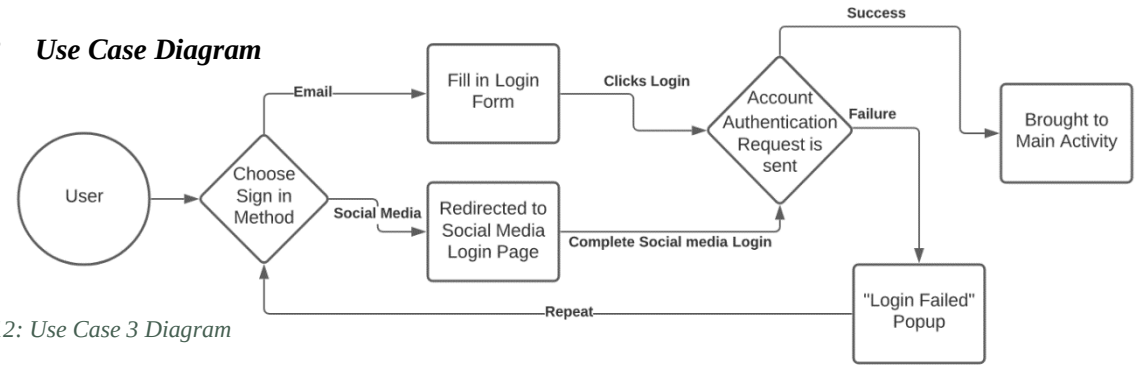

Figure 12: Use Case 3 Diagram

3.3.3.4 Precondition

The user has started the application and has previously set up an account but is not currently logged in.

3.3.3.5 Activation

The system detects that the user is not logged in.

3.3.3.6 Main Flow

Actor Action:

1. User fills in the required information: email, password.
2. The user clicks the “Log In” button.

System Response: Account log in request is sent to the firebase authentication service. Upon success, the user is logged in. Upon failure and “Login Failed” snackbar notification appears.

3.3.3.7 Alternate Flow

Actor Action:

1. User clicks the chosen social media log in method. (Twitter, Google, Facebook)
2. The user is redirected to the login of chosen social media.
3. User completes selected login method

System Response: Account log in request is sent to the firebase authentication service. Upon success, the user is logged in. Upon failure and “Login Failed” snackbar notification appears.

3.3.3.8 Terminating

When account authentication is successful, this flow is terminated.

3.3.3.9 Post Condition

The user is brought to the main application screen.

3.3.4 Use Case 4: Create New Expense

3.3.4.1 Description & Priority
This use case is fundamental to the basic functionality of the application.

3.3.4.2 Scope
The scope of this use case is to allow a user to create an expense to be split among users.

3.3.4.3 Use Case Diagram

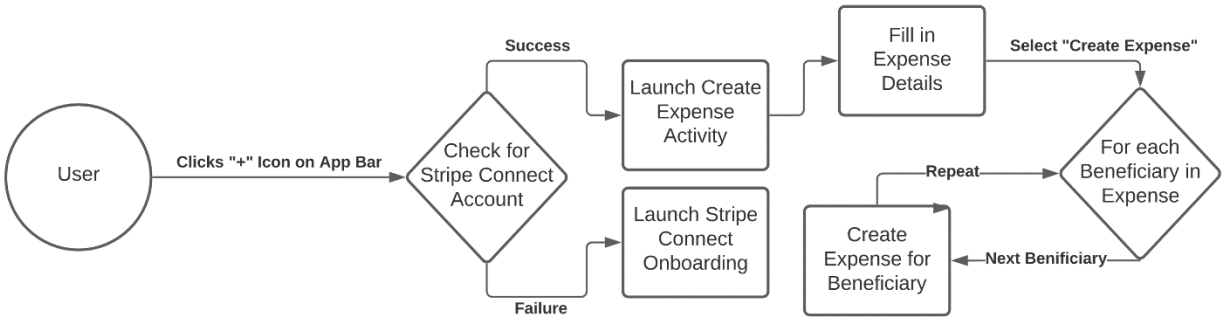

Figure 13: Use Case 4 Diagram

3.3.4.4 Precondition
User has already set up a Stripe Connect account and linked their preferred bank account.

3.3.4.5 Activation
User selects the new expense “+” icon.

3.3.4.6 Main Flow
Actor Action:

1. User fills in the required information: name, price, includes yourself (checkbox)
2. User selects beneficiaries.
3. The user clicks the “Create Expense” button.

System Response: For each beneficiary indicated, a document is added to the incoming collection within the database document of the benefactor. When a document is added to this collection, a cloud function is triggered which registers the required information to stripe and places a similar document in the due collection within the database document of the beneficiary. This process is repeated until payments have been created for all beneficiaries.

3.3.4.7 Alternate Flow
System Response: The user is redirected to the Stripe onboarding activity and prompted to create a stripe connect account.

3.3.4.8 Terminating
When expense has been created for each beneficiary, this flow is terminated.

3.3.4.9 Post Condition
The user is brought to the main application screen.

3.3.4.10 Full Use Case Diagram

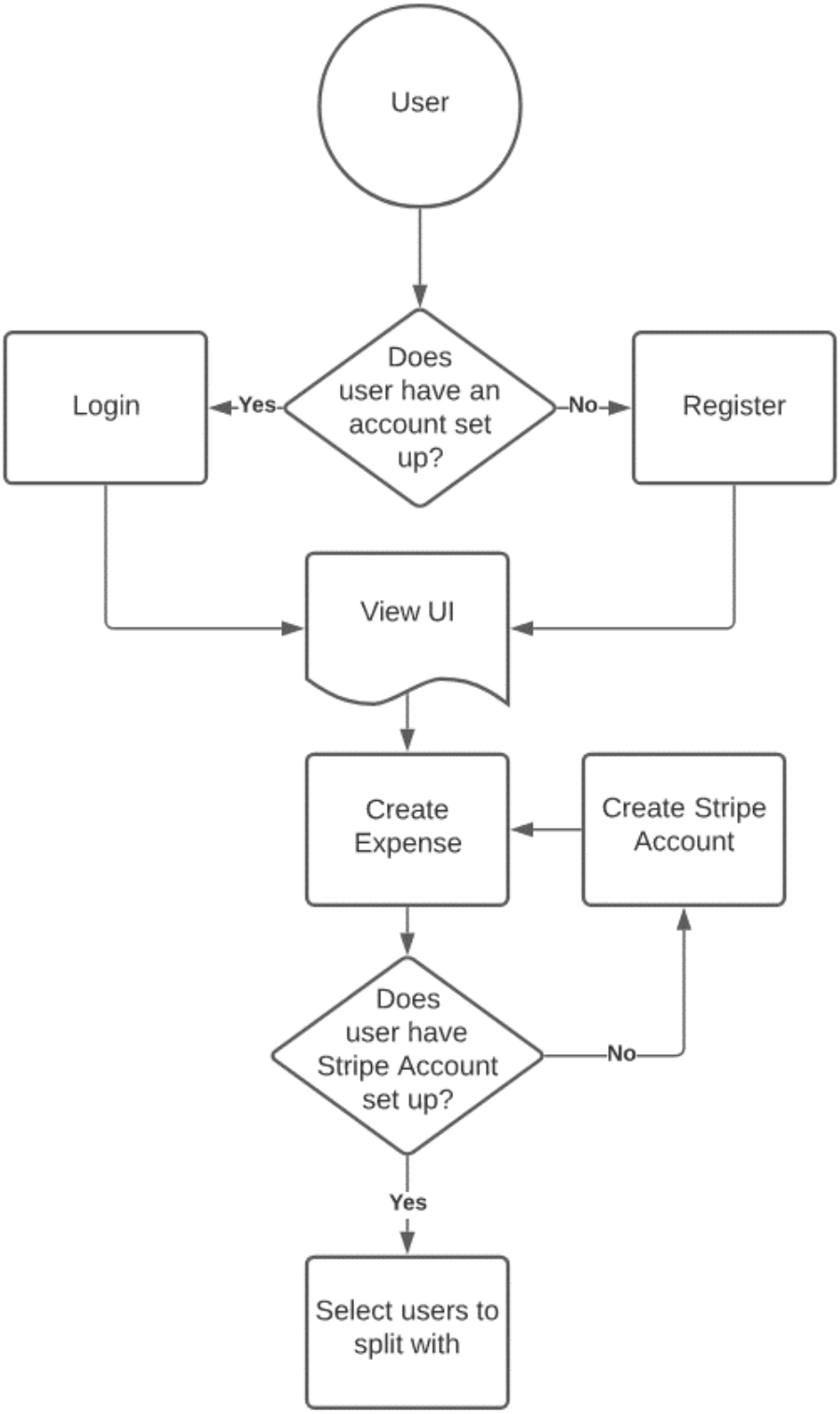

Figure 14: Create Expense Full Flow

3.3.5 Use Case 5: Create Stripe Account

3.3.5.1 Description & Priority

This use case is required for the functionality of the stripe connect API

3.3.5.2 Scope

The scope of this use case is to create a stripe connect account within the application using the stripe connect onboarding functionality.

3.3.5.3 Use Case Diagram

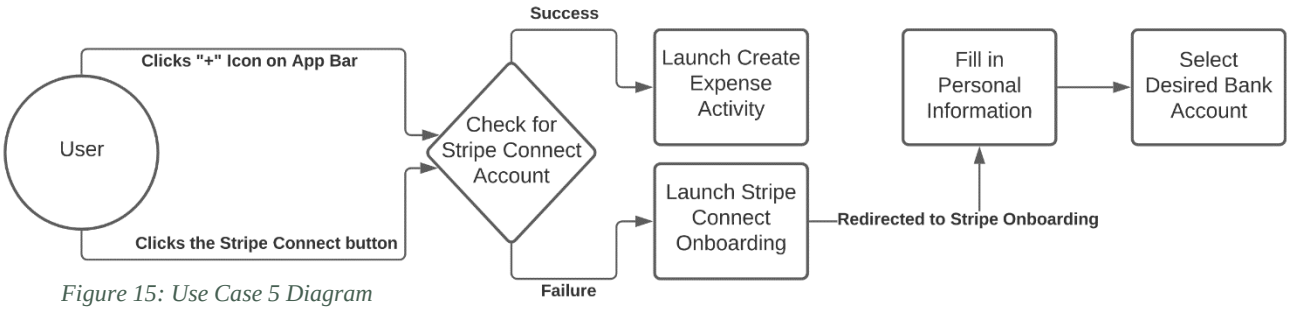

Figure 15: Use Case 5 Diagram

3.3.5.4 Precondition

The user has started the application and has previously set up an account but is not currently logged in.

3.3.5.5 Activation

The system detects that the user has not created an account when:

1. User attempts to create an expense.

   or

2. User selects Stripe Connect button in menu activity.

3.3.5.6 Main Flow

Actor Action:

1. The user fills in the required personal information.
2. The user fills in account information for the desired bank account.

System Response: User is redirected to stripe onboarding activity and directed to complete their application through stripe API.

3.3.5.7 Terminating

When the user has created their stripe connect account, this flow is terminated.

3.3.5.8 Post Condition

The user is brought to the main application activity.

Use Case 6: Pay an Expense

3.3.5.9 Description & Priority
This use case facilitates the payment functionality of the application.

3.3.5.10 Scope
The scope of this use case is to pay for expenses with available payment methods.

3.3.5.11 Use Case Diagram

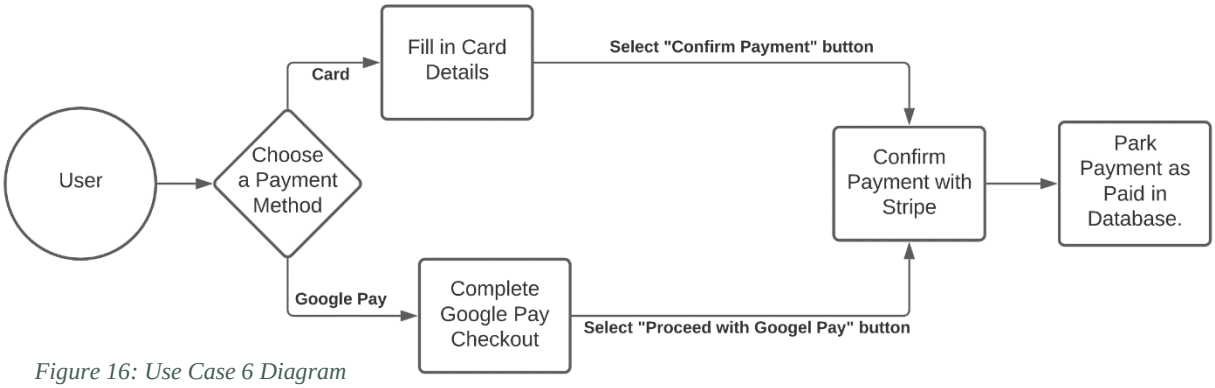

Figure 16: Use Case 6 Diagram

3.3.5.12 Precondition
The user has started the application and has previously set up an account but is not currently logged in.

3.3.5.13 Activation
User selects the new expense “Pay Up” icon of a particular payment in the due fragment.

3.3.5.14 Main Flow
Actor Action:

1. User selects “Pay with Google Pay”.
2. The user is shown a Google Pay overlay.
3. Complete Google Pay payment.

System Response: Google Pay payment method is passed to stripe through confirm payment method in Stripe API.

3.3.5.15 Alternate Flow
Actor Action:

1. User fills in card details.
2. User selects “Confirm Payment”.

System Response: Card payment method is passed to stripe through confirm payment method in Stripe API.

3.3.5.16 Terminating
When payment is completed successfully, this flow is terminated.

3.3.5.17 Post Condition
The user is brought to the main application screen.

3.3.5.18 Full Use Case Diagram

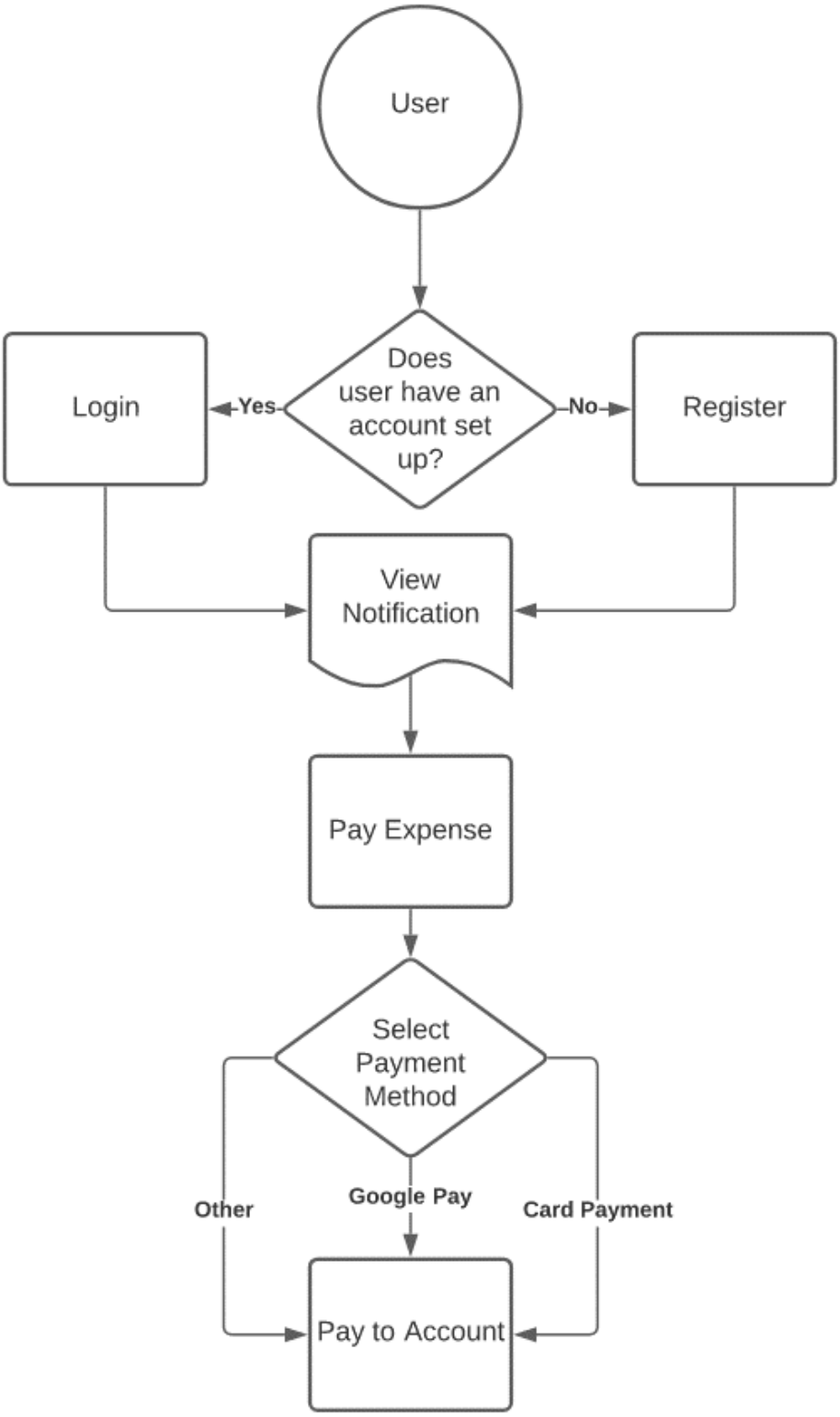

Figure 17: Pay Expense Full Flow

4 IMPLEMENTATION

4.1 ANDROID

Permissions are significant factor found in the Android Manifest configuration file. This is needed if the application is involved in accessing external functions. For example, whether the application requires access to the internet, the camera, or some feature, it needs permissions. It exists as an XML file. Permissions are automatically created for simple programs at the time of application initialisation. If the app uses a higher-level API or SDK, the permissions within the app must be clearly stated as a permissions tag for accessing functions or components. Below are the additional permissions required for application functionality:

1. android.permission.INTERNET
2. android.permission.ACCESS_NETWORK_STATE

These are used for accessing the internet and checking for internet connection state respectively.

4.2 FIRESTORE DATABASE

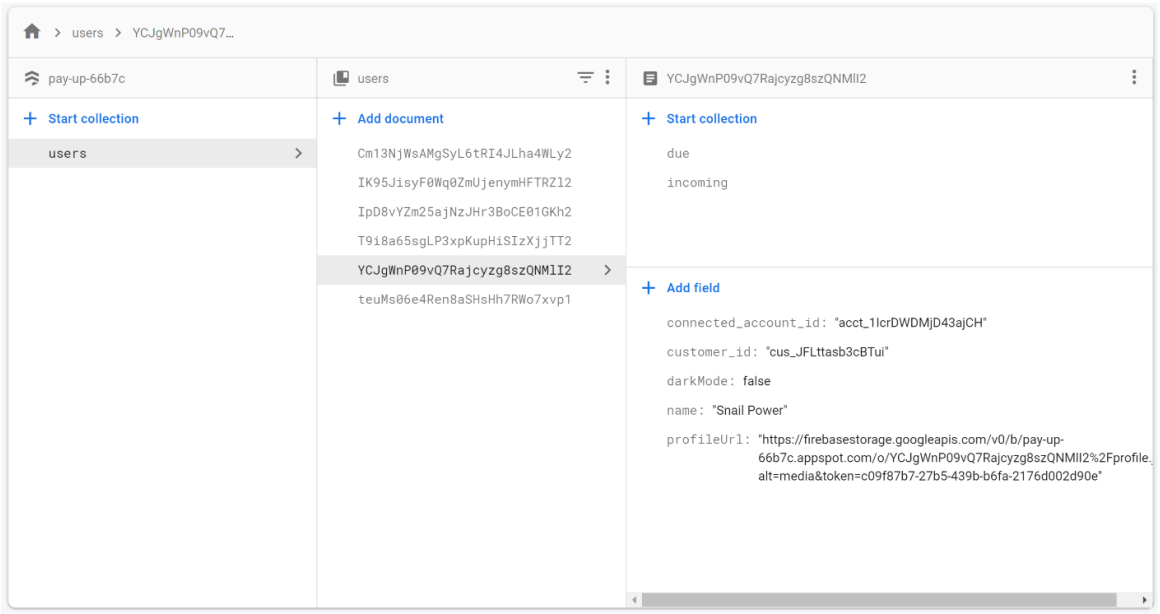

Figure 18: Cloud Firestore

The database implemented for this project is the Firebase Cloud Firestore. It is a document-based database. The user's settings, stripe account details and customer details are stored in this document.

I structured the database to contain all payments within the user document. Each user document has two additional sub-collections, “incoming” and “due”. When a payment is created it is added to the “incoming” collection of the user who created the expense. This then triggers a cloud function that creates the stripe payment and copies the expense document to the due sub-collection of the beneficiary.

4.3 CLOUD FUNCTIONS

For the automated backend server features, I chose to use the Google Cloud Platform’s Cloud Functions. These functions sync with the firebase project and allow for automated function calls on database logging and authentication of users.

This server-side of the project is written in entirely Node 10 JavaScript. The first thing that needed to be done was syncing the Firebase project with the selected functions. This can be done with the command line upon installing the Firebase CLI.

As these functions are crucial to the functionality of the application. I will detail the functions here and refer back to them later in the document.

The functions implemented are as follows:

4.3.1 CreateUserDocument

```
/**
 * When a user is created, create a Stripe customer object for them and add user to database.
 */
exports.createUserDocument = functions.auth.user().onCreate(async (user) => {
  const customer = await stripe.customers.create({
    email: user.email,
    metadata: { firebaseUID: user.uid },
  });

  //add to database
  await admin.firestore().collection('users').doc(user.uid).set({
    customer_id: customer.id,
    name: user.displayName,
  });

  return;
});
```

Figure 19: CreateUserDocument

This function is called on the creation of a new user. Once the issuer is authorized the user information is passed to this function which creates the stripe customer account (separate to a Stripe connected account) and places the basic user information into the database with a document id equal to the user uid. The stripe customer account number is also saved to be used during checkout.

4.3.2 AuthorizeStripeAccount

```
/**
 * Autherize a stripe account during onboarding
 */
exports.authorizeStripeAccount = functions.https.onCall(async (data, context) => {
  // Checking that the user is authenticated.
  if (!context.auth) {
    // Throwing an HttpsError so that the client gets the error details.
    throw new functions.https.HttpsError(
      'failed-precondition',
      'The function must be called while authenticated!'
    );
  }
  const uid = context.auth.uid;
  try {
    if (!uid) throw new Error('Not authenticated!');
    // Get stripe customer id
    const code = data.code;
    const response = await stripe.oauth.token({
      grant_type: 'authorization_code',
      code: code,
    });

    var connected_account_id = response.stripe_user_id;

    await admin.firestore().collection('users').doc(uid).update({
      connected_account_id: connected_account_id,
    });
    return "Complete";
  } catch (error) {
    throw new functions.https.HttpsError('internal', error.message);
  }
});
```

Figure 20: AuthorizeStripeAccount

This function is called manually by the application when a new user completes the Stripe onboarding process within the application. User data is sent in the form of context and the required authorization code is sent through the “data” field of the request. The authorization code is sent to stripe through an oath request, and upon completion, the connected account id is returned to the function. This account id is then saved to the user document. This account id is required to set the target account for any payments created by this user.

4.3.3 CreateStripePayment

```
/**
 * When a payment document is written on the client,
 * this function is triggered to create the PaymentIntent in Stripe.
 */
exports.createStripePayment = functions.firestore
  .document('users/{userId}/incoming/{pushId}')
  .onCreate(async (snap, context) => {
    const { amount, currency, user_id, service_name, date_created} = snap.data();
    try {
      // Look up the Stripe customer id.
      const customer = (await snap.ref.parent.parent.get()).data().connected_account_id;
      const username = (await snap.ref.parent.parent.get()).data().name;
      // Create a charge using the pushId as the idempotency key to protect against double charges.
      const idempotencyKey = context.params.pushId;
      const payment = await stripe.paymentIntents.create(
        {
          payment_method_types: ['card', ],
          amount: amount,
          currency: currency,
          on_behalf_of: customer,
          transfer_data: {
            destination: customer,
          },
        });
      const clientSecret = payment.client_secret;
      // If the result is successful, write it back to the database.
      //await snap.ref.set(payment);

      await admin.firestore().collection('users').doc(user_id).collection('due').doc(idempotencyKey).set({
        service_name: service_name,
        amount: amount,
        clientSecret: clientSecret,
        currency: currency,
        date_created: date_created,
        user_id: context.params.userId,
        user_name: username,
        active: true
      });

    } catch (error) {
      // We want to capture errors and render them in a user-friendly way,
      //while still logging an exception with StackDriver
      console.log(error);
      await snap.ref.set({ error: userFacingMessage(error) }, { merge: true });
      await reportError(error, { user: context.params.userId });
    }
  });
```

Figure 21: CreateStripePayment

This function is used to create the stripe payment intent object. It is called when a new document is added to the incoming subcollection of the user. The basic information for the payment is then extracted from the document and sent to stripe to create a payment intent the “transfer_data: destination” refers to the connected account of the benefactor of the expense. This is gotten from the user document in which the payment details were added. The client secret is used during checkout when completing the payment.

The stripe service then provides a client secret. This secret is stored back in the document. A copy of the payment document is saved to the due subcollection of the beneficiary of the expense. The only change made to the document Is the user id and username. For the benefactor, these values represent the beneficiary. For beneficiary the benefactor, these values represent the benefactor.

4.3.4 MarkasPaid

```
/**
 * When a user pays an Expense, mark the expense as paid for the benefactor
 */
 exports.markAsPaid = functions.firestore
   .document('users/{userId}/due/{pushId}')
   .onUpdate(async (change, context) => {
     const newValue = change.after.data();
     const active = newValue.active;
     const date_paid = newValue.date_paid;
     const payment_method = newValue.payment_method;
     const user_id = newValue.user_id;
     try {
       const idempotencyKey = context.params.pushId;

       await admin.firestore().collection('users').doc(user_id).collection('incoming').doc(idempotencyKey).update({
         active: active,
         date_paid: date_paid,
         payment_method: payment_method
       });

     } catch (error) {
       // We want to capture errors and render them in a user-friendly way, while still logging an exception with StackDriver
       console.log(error);
       await snap.ref.set({ error: userFacingMessage(error) }, { merge: true });
       await reportError(error, { user: context.params.userId });
     }
   });
```

Figure 22: MarkasPaid

This function is activated when a payment document in the due subcollection of any user is changed. It takes the change data and applies it to the corresponding payment document in the incoming subcollection of the benefactor. This happens when checkout is completed. Active is set to false, the date paid is recorded, and the payment method is recorded.

4.3.5 MarkasRecieved

```
/**
 * When a Benefactor marks an Expense as paid, mark the expense as paid for the benefactor
 */
 exports.markAsReceived = functions.firestore
 .document('users/{userId}/incoming/{pushId}')
 .onUpdate(async (change, context) => {
   const newValue = change.after.data();
   const active = newValue.active;
   const date_paid = newValue.date_paid;
   const payment_method = newValue.payment_method;
   const user_id = newValue.user_id;
   try {
     const idempotencyKey = context.params.pushId;

     await admin.firestore().collection('users').doc(user_id).collection('due').doc(idempotencyKey).update({
       active: active,
       date_paid: date_paid,
       payment_method: payment_method
     });

   } catch (error) {
     // We want to capture errors and render them in a user-friendly way, while still logging an exception with StackDriver
     console.log(error);
     await snap.ref.set({ error: userFacingMessage(error) }, { merge: true });
     await reportError(error, { user: context.params.userId });
   }
 });
```

Figure 23:MarkasRecieved

This function is activated when a payment document in the incoming subcollection of any user is changed. It takes the change data and applies it to the corresponding payment document in the due subcollection of the benefactor. This happens when the benefactor marks a payment as received. The payment method is set as “Outside of App”.

4.3.6 CleanUpUser

```
/**
 * When a user deletes their account, clean up after them.
 */
exports.cleanupUser = functions.auth.user().onDelete(async (user) => {
  const dbRef = admin.firestore().collection('users');
  const customer = (await dbRef.doc(user.uid).get()).data();
  await stripe.customers.del(customer.customer_id);
  // Delete the customers payments & payment methods in firestore.
  const snapshot = await dbRef
    .doc(user.uid)
    .collection('payment_methods')
    .get();
  snapshot.forEach((snap) => snap.ref.delete());

  //delete any paymetns incoming
  try {
    const incomingSnapshot = await dbRef
      .doc(user.uid)
      .collection('incoming')
      .get();
    incomingSnapshot.forEach((snap) => {
      snap.ref.delete();
    });
  } catch (error) {
    console.log(error);
  }

  //delete any paymetns due
  try {
    const dueSnapshot = await dbRef
      .doc(user.uid)
      .collection('due')
      .get();
    dueSnapshot.forEach((snap) => {
      snap.ref.delete();
    });
  } catch (error) {
    console.log(error);
  }

  //delete user document
  await dbRef.doc(user.uid).delete();

  //delete the connected account attached to this user
  const connected_account_id = customer.connected_account_id
  await stripe.oauth.deauthorize({
    client_id: '[redacted]',
    stripe_user_id: connected_account_id,
  });
});
```

Figure 24:CleanUpUser

This function is activated when a user is removed through the authentication tab of the Firebase console. This function deletes all payments relating to that user in both due and incoming. These deletions trigger the DueDeleted and IncomingDeleted functions respectively. The function then deauthorises the customer with stripe and deletes the user document from the database.

4.3.7 IncomingDeleted

```
/**
* When a incoming payment is deleted it is removed from the corrisponding due
*/
exports.incomingDeleted = functions.firestore
.document('users/{userId}/incoming/{pushId}')
.onDelete(async (snap, context) => {
  try {
    const push_id = context.params.pushId;
    const user_id = snap.data().user_id;
    await admin.firestore().collection('users').doc(user_id).collection('due').doc(push_id).delete();
  } catch (error) {
    console.log(error);
    await reportError(error, { user: context.params.userId });
  }
});
```

Figure 25: Incoming Deleted

When an incoming document is deleted, whether by the server admin or the cleanUpUser function. This function will then delete the corresponding due document of the beneficiary.

4.3.8 DueDeleted

```
/**
* When a due payment is deleted it is removed from the corrisponding incoming
*/
exports.dueDeleted = functions.firestore
.document('users/{userId}/due/{pushId}')
.onDelete(async (snap, context) => {
  try {
    const push_id = context.params.pushId;
    const user_id = snap.data().user_id;
    await admin.firestore().collection('users').doc(user_id).collection('incoming').doc(push_id).delete();
  } catch (error) {
    console.log(error);
    await reportError(error, { user: context.params.userId });
  }
});
```

Figure 26: DueDeleted

When a due document is deleted, whether by the server admin or the cleanUpUser function. This function will then delete the corresponding incoming document of the benefactor.

4.3.9 ReportError

```
function reportError(err, context = {}) {
  /**
  * This is the name of the StackDriver log stream that will receive the log
  * entry. This name can be any valid log stream name, but must contain "err"
  * in order for the error to be picked up by StackDriver Error Reporting.
  */
  const logName = 'errors';
  const log = logging.log(logName);

  const metadata = {
    resource: {
      type: 'cloud_function',
      labels: { function_name: process.env.FUNCTION_NAME },
    },
  };

  const errorEvent = {
    message: err.stack,
    serviceContext: {
      service: process.env.FUNCTION_NAME,
      resourceType: 'cloud_function',
    },
    context: context,
  };

  // Write the error log entry
  return new Promise((resolve, reject) => {
    log.write(log.entry(metadata, errorEvent), (error) => {
      if (error) {
        return reject(error);
      }
      return resolve();
    });
  });
}
```

Figure 27: Report Error

This function is simply used to report errors to the logs in a more readable fashion. This code was provided by the official stripe “Android basic integration”. (Stripe, n.d.)

4.3.10 UserFacingMessage

```
/**
 * Sanitize the error message for the user.
 */
function userFacingMessage(error) {
  return error.type
    ? error.message
    : 'An error occurred, developers have been alerted';
}
```

Figure 28: UserFacingMessage

This function places a copy of the error in the document during the processing of that the error occurred. This is useful when viewing a large amount of payment to determine where the error occurred.

4.3.11 CreateEphemeralKey

```
/**
 * Set up an ephemeral key.
 */
exports.createEphemeralKey = functions.https.onCall(async (data, context) => {
  // Checking that the user is authenticated.
  if (!context.auth) {
    // Throwing an HttpsError so that the client gets the error details.
    throw new functions.https.HttpsError(
      'failed-precondition',
      'The function must be called while authenticated!'
    );
  }
  const uid = context.auth.uid;
  try {
    if (!uid) throw new Error('Not authenticated!');
    // Get stripe customer id
    const customer = (
      await admin.firestore().collection('users').doc(uid).get()
    ).data().customer_id;
    const key = await stripe.ephemeralKeys.create(
      { customer },
      { apiVersion: data.api_version }
    );
    return key;
  } catch (error) {
    throw new functions.https.HttpsError('internal', error.message);
  }
});
```

Figure 29: CreateEphemeralKey

This function is currently not active but will be necessary when storing payment methods for later user (See Further Development, Save Payment Details.Recurring Payments)

4.4 FIREBASE AUTHENTICATION

All login and registrations are implemented into SignIn.java class. A Firebase Auth listener is used to detect a successful login. During Authentication Previously attempted to check for auth success at the end of each login methods. But due to an authentication delay, this required multiple successive sign-ins. The auth listener improved sign-in performance significantly.

```
//FireAuth initialise
mAuth = FirebaseAuth.getInstance();
//Auth listener to check for successful login
FirebaseAuth.AuthStateListener authStateListener = firebaseAuth -> {
    if (mAuth.getCurrentUser() != null){
        setResult(RESULT_OK);
        finish();
    }
};
mAuth.addAuthStateListener(authStateListener);
```

Figure 30: checkCurrentUser

Only when a user has successfully signed is this activated. Upon confirming sign in the SignIn activity is finished. In the main activity, there is a similar listener which will return the user to the sign in if user upon sign out. A progress bar appears, and all buttons and fields are rendered inaccessible.

Upon return to the main activity, the user's information is received with the getUserProfile method. This method will return creates a CurrentUser object with all the relevant user information that is saved to the database for the authenticated user.

4.4.1 Email

4.4.1.1 Login

For the login, I created two editText fields, email and password, and a login button. The email and password are taken from the editText fields. When the email button is pressed, these two strings are passed into an email login method loginSignIn(emailString, passwordString).

In this method, the strings are first checked for errors using validateLoginInForm().

```
//validate login strings correctness
private boolean validateLoginInForm(String emailString, String passwordString) {
    //check email correctness
    if (emailString.isEmpty() || !android.util.Patterns.EMAIL_ADDRESS.matcher(emailString).matches()) {
        emailInput.setError("Required.");
        return true;
    } else {
        emailInput.setError(null);
        progressBar.setVisibility(View.INVISIBLE);
    }

    //check password correctness
    if (passwordString.isEmpty()) {
        passwordInput.setError("Required.");
        return true;
    } else {
        passwordInput.setError(null);
        progressBar.setVisibility(View.INVISIBLE);
    }
    return false;
}
```

Figure 31: ValidateLogInForm

Upon validation success, Firebase authenticator is called, and the strings are passed into signInWithEmailAndPassword(email, password). This task either returns a successful login or fails.

- On successful login, the auth listener is called, the current user is checked, and "signInWithEmail:success" is saved to the log.
- On failed login, a snackbar popup appears with the message “Authentication Failed”. “signInWithEmail:failure” saved to log along with exception details.

4.4.1.2 Registration

For registration, there is a “No Account Yet? Create one”. This will then hide the social media buttons and a login button. This also reveals a “Register” button. The previous email and password fields also remain. The 2 strings name and password fields are then passed into the createAccount method.

This method then validated these 4 strings within validateLoginInForm() (See Figure 31: ValidateLogInForm)

Upon validation success, Firebase authenticator is called, and the strings are passed into CreateUserWithEmailAndPassword. This task either returns a successful login or fails.

- the auth listener is called, the current user is checked, and "signInWithEmail:success" is saved to the log.
- On failed login, a snackbar popup appears with the message “Registration Failed”. “createUserWithEmail:failure” saved to log along with exception details.

4.4.2 Google

In onCreate(), a google sign in options is created and passed into a googleSignInClient. This client is used to create the sign-in intent which is passed into a googleActivityResultLauncher which handles the result.

```
//handle google activity result
ActivityResultLauncher<Intent> googleActivityResultLauncher = registerForActivityResult(
        new ActivityResultContracts.StartActivityForResult(),
        result -> {
            if (result.getResultCode() == Activity.RESULT_OK) {
                Intent data = result.getData();
                Task<GoogleSignInAccount> task = GoogleSignIn.getSignedInAccountFromIntent(data);
                try {
                    // Google Sign In was successful, authenticate with Firebase
                    GoogleSignInAccount account = task.getResult(ApiException.class);
                    assert account != null;
                    Log.d(TAG,  msg: "firebaseAuthWithGoogle:" + account.getId());
                    firebaseAuthWithGoogle(account.getIdToken());
                } catch (ApiException e) {
                    // Google Sign In failed, update UI appropriately
                    Log.w(TAG,  msg: "Google sign in failed", e);
                    Snackbar.make(findViewById(android.R.id.content),  text: "Authentication Failed.", Snackbar.LENGTH_LONG)
                            .setAction( text: "Action",  listener: null).show();
                }
            }
        });
```

Figure 32: onActivityResult

This method will then request user data using the result data. This task will return either a success or failure.

- On success, account info is received and firebaseAuthWithGoogle is called using the account id token. "firebaseAuthWithGoogle:" followed by the account id is added to the log.
- On failure, snackbar popup indicates “Authentication Failure”. “Google sign in failed” is added to log along with exception details.

```
//handle google authentication
private void firebaseAuthWithGoogle(String idToken) {
    AuthCredential credential = GoogleAuthProvider.getCredential(idToken,  s1: null);
    socialAuthentication(credential);
}
```

Figure 33: firebaseAuthWithGoogle

firebaseAuthWithGoogle gets the credential for the account using the id token. This credential is then passed to the socialAuthentication function.

```
//authenticate social accesstoken
private void socialAuthentication(AuthCredential credential) {
    mAuth.signInWithCredential(credential)
            .addOnCompleteListener( activity: this, task -> {
                if (task.isSuccessful()) {
                    // Sign in success, update UI with the signed-in user's information
                    Log.d(TAG,  msg: "signInWithCredential:success");
                    setResult(RESULT_OK);
                } else {
                    // If sign in fails, display a message to the user.
                    Log.w(TAG,  msg: "signInWithCredential:failure", task.getException());
                    Snackbar.make(findViewById(android.R.id.content),  text: "Authentication Failed.", Snackbar.LENGTH_LONG)
                            .setAction( text: "Action",  listener: null).show();
                    progressBar.setVisibility(View.INVISIBLE);
                }
            });
```

Figure 34: Social Authentication

The social authentication method calls Firestore Authenticator with the signInWithCredential method. This method functions the same as the signInWithEmailAndPassword method used for email login. The results of the task are similar for both.

When signing in or registering with google, a google popup will appear allowing the user to select a Google account which they have saved to their device. Once the user has selected an account once, this popup will no longer appear

4.4.3 Facebook

Facebook required additional steps when compared to the Google sign in.

The first requirement is the creation of a Facebook developers account. This was a simple enough task and Facebook provide the most up to date *includes* and *Manifest Permissions* required to run their API.

Facebook also provide a pre-configured login button which is used to create the sign-in intent. But this button could not be modified and as such, I choose to use a different method.

The method I choose was to create a login manager with reading permissions for public profile, email and name and included a mCallbackManagaer into the login manager.

```
mCallbackManager = CallbackManager.Factory.create();
LoginManager.getInstance().registerCallback(mCallbackManager, new FacebookCallback<LoginResult>() {
    @Override
    public void onSuccess(LoginResult loginResult) {
        Log.d(TAG,  msg: "facebook:onSuccess:" + loginResult);
        handleFacebookAccessToken(loginResult.getAccessToken());
    }
    @Override
    public void onCancel() {
        Log.d(TAG,  msg: "facebook:onCancel");
        Snackbar.make(findViewById(android.R.id.content),  text: "Authentication Cancelled.", Snackbar.LENGTH_LONG)
                .setAction( text: "Action",  listener: null).show();
        progressBar.setVisibility(View.INVISIBLE);
    }
    @Override
    public void onError(FacebookException error) {
        Log.d(TAG,  msg: "facebook:onError", error);
        Snackbar.make(findViewById(android.R.id.content),  text: "Authentication Failed.", Snackbar.LENGTH_LONG)
                .setAction( text: "Action",  listener: null).show();
        progressBar.setVisibility(View.INVISIBLE);
    }
});
```

Figure 35: Callback Manager

This callback manager was then placed into the onActivityResult. Upon clicking the new custom Facebook login button. This callback manager is called and a login result containing an access token is received. This Facebook access token is then passed to the handleFacebookAccessToken which functions identically as the firebaseAuthWithGoogle indicated in Figure 33. The only differentiation is the use of FacebookAuthProvider instead of the GoogleAuthProvider.

4.6 CLOUD STORAGE

Cloud storage is used solely to store the profile pictures of the users. These pictures can be uploaded from the User Menu of the application and only a single photo is uploaded by the user. Once a user changes their photo the original photo is replaced. The photos are stored in https://console.firebase.google.com/project/pay-up-66b7c/storage/pay-up-66b7c.appspot.com/files under the file scheme "uid/profile.jpg"

4.7 STRIPE CONNECT

4.7.1 Connected Account Setup

Stripe Connected account onboarding is implemented through the Oauth redirect URL provided by Stripe. This URL is loaded into a WebView inside of the Stripe Onboarding activity. Once the user completes the Stripe onboarding through his embedding form. An authorisation code is returned at the end of a redirect URL. For the redirect URL, the location is meant to represent an application splash page or company site. As I do not have either of these, I simply set the redirect URL to my GitHub page. Once a redirect to this page is detected, the code is extracted and the AuthorizeStripeAccount cloud function. The activity is then ended, and the state of the connected account is updated.

4.7.2 Create Payment

A user cannot create an expense until they have completed the Stripe onboarding process. If they attempt to access the create Expense activity without completing this, they will be redirected to Stripe onboarding.

When creating a payment, the user enters in the name of the expense and the total cost of the expense. The user can also select whether this cost includes their portion of the payment or not. An on-text change listener is set so that as the price is entered it is automatically formatted to the currency of the user and split among the selected beneficiaries.

```
private void formatPrice() {
    //get price
    String s = Objects.requireNonNull(priceInput.getEditText()).getText().toString();
    String cleanString = s.replaceAll( regex: "[$,£€.]",  replacement: "");

    //parse string
    double parsed = Double.parseDouble(cleanString);
    perPerson = parsed;

    //convert to selected currency
    pricePP.setText(String.format("%s%s", "Price per person:", NumberFormat.getCurrencyInstance().format((perPerson/ 100 / (addedUsers.size() + includes)))));

    String formatted = NumberFormat.getCurrencyInstance().format((parsed/100));

    //set the new formatted value
    current = formatted;
    priceInput.getEditText().setText(formatted);
    priceInput.getEditText().setSelection(formatted.length());
}
```

Figure 36: Format Price

The user is selected through a recycler view of user objects. These objects appear as a card with the user's name, profile picture and unhighlighted star. When a user object is clicked this star is highlighted and the user is added to the addedUsers List.

When the “Create Expense” button is clicked, the form fields are validated for the correctness and a loop is initialised which creates a payment object in the incoming subcollection of the benefactor for each beneficiary in the addedUsers list.

```
//get prcie per person
long price = Math.round((perPerson / (addedUsers.size() + includes)));

//check for valid price
if (price >= 50 && addedUsers.size() != 0) {
    //for each of the added user
    for (int i = 0; i < addedUsers.size(); i++) {
        User user = addedUsers.get(i);
        String uid = user.getId();
        String name = user.getUsername();
        String amount = String.valueOf(price);

        //get current dateTime
        @SuppressLint("SimpleDateFormat") SimpleDateFormat sdf = new SimpleDateFormat( pattern: "dd/MM/yyyy\nHH:mm z");
        String currentDateAndTime = sdf.format(new Date());

        //create the payment data for teh document
        Map<String, Object> paymentDetails = new HashMap<>();
        paymentDetails.put( k: "user_id", uid);
        paymentDetails.put( k: "user_name", name);
        paymentDetails.put( k: "currency", currency.getCurrencyCode());
        paymentDetails.put( k: "amount", amount);
        paymentDetails.put( k: "service_name", serviceName);
        paymentDetails.put( k: "active",  v: true);
        paymentDetails.put( k: "date_created", currentDateAndTime);

        //add document to current users incoming collection
        db.collection( collectionPath: "users").document(currentUser.getId()).collection( collectionPath: "incoming") CollectionReference
                .document() DocumentReference
                .set(paymentDetails) Task<Void>
                .addOnSuccessListener(aVoid -> Log.d(TAG,  msg: "DocumentSnapshot successfully written!"))
                .addOnFailureListener(e -> Log.w(TAG,  msg: "Error writing document", e));
    }
    //once all documents are created end activity
    finish();
} else {
    //popup to indicate invalid price
    Snackbar.make(findViewById(android.R.id.content),  text: "Price per person must be at least 0.50 and at least one user must be selected.", Snackbar.LENGTH_LONG)
            .setAction( text: "Action",  listener: null).show();
    progressBar.setVisibility(View.INVISIBLE);
}
```

Figure 37: Create Group Expense

Each payment document created triggers the CreateStripePayment function. Which then initialises the payment intent with stripe and creates the corresponding document in the due sub-collections of the beneficiaries. If the price per person is less than 0.50 EUR, then a popup appears to indicate this to the user and no payment is created.

4.7.3 Checkout

The checkout activity is written in Koitlin. I had never previously used Koitlin but wished to attempt to utilise it. My Main reason why I chose to create this activity in Koitlin instead of any other, is that there was a greater amount of online resources for Koitlin.

There are two methods for payment. The first is a card input widget that allows the user to input their desired card details and confirm the payment through this.

```
private fun confirmPayment() {
    val params = binding.cardInputWidget.paymentMethodCreateParams
    if (params != null) {
        val confirmParams = ConfirmPaymentIntentParams
                .createWithPaymentMethodCreateParams(params, clientSecret)
        stripe.confirmPayment( activity: this, confirmParams)
    }
}
```

Figure 38: Confirm Payment

When the user confirms the payment, the payment method object is created from the card input widget and passed into the confirm payment stripe function of the stripe API.

The on result for this confirm payment simply prints a popup that simply displays the payment success state and sets the return intent with the payment method used.

4.7.4 Google Pay

The Google Pay method was more complicated to implement than the card payment. First, I check to see if Google pay is possible. This is done using the isReadyToPay method which sends an HTTP request to google payment client. If this returns a successful request, the Google Pay button is enabled. Upon clicking this button, a Google Pay overlay is created to collect the user data and run the two-factor authentication. These payment details are then sent to the Goole Pay client that then returns the formatted payment method to the on-activity result. This is then redirected to the onGoogelPayResult

```
private fun onGooglePayResult(data: Intent) {
    val paymentData = PaymentData.getFromIntent(data) ?: return
    val paymentMethodCreateParams =
            PaymentMethodCreateParams.createFromGooglePay(
                    JSONObject(paymentData.toJson())
            )

    // now use the `paymentMethodCreateParams` object to create a PaymentMethod
    stripe.createPaymentMethod(
            paymentMethodCreateParams,
            callback = object : ApiResultCallback<PaymentMethod> {
                override fun onSuccess(result: PaymentMethod) {
                    val confirmParams = ConfirmPaymentIntentParams
                            .createWithPaymentMethodId(result.id!!, clientSecret)
                    stripe.confirmPayment( activity: this@CheckoutActivity, confirmParams)
                }

                override fun onError(e: Exception) {
                }
            }
    )
```

Figure 39: On Google Pay Result

This then confirms the payment with the stripe API in the same way that the card confirms payment.

4.8 USER SETTINGS

4.8.1 Profile Picture

```
//open gallery to select profile image
private void openGallery() {
    Intent openGalleryIntent = new Intent(Intent.ACTION_PICK, MediaStore.Images.Media.EXTERNAL_CONTENT_URI);
    galleryResultLauncher.launch(openGalleryIntent);
}

//activity handler for gallery
ActivityResultLauncher<Intent> galleryResultLauncher = registerForActivityResult(
        new ActivityResultContracts.StartActivityForResult(),
        result -> {
            if (result.getResultCode() == RESULT_OK) {
                Intent data = result.getData();
                if (data != null) {
                    //set teh new image
                    Uri image = data.getData();
                    uploadImage(image);
                }
            }
        });
```

Figure 40: Open Gallery

When a user clicks on their profile picture, media store image intent is created. This appears to the user as an overlay listing all gallery applications on their phones. Once a preferred application is selected the user is redirected to that application. Upon selecting an image, the user is returned to the application and the local image Uri is passed to the uploadImage() method.

```
private void uploadImage(Uri image) {
    //set storage refrenece for profile image of current user
    StorageReference fileRef = mStorageRef.child(currentUser.getId() + "/profile.jpg");
    //upload new file
    fileRef.putFile(image).addOnSuccessListener(taskSnapshot -> fileRef.getDownloadUrl().addOnSuccessListener(uri -> {
        //set user object to new profile uri in firebase storage
        currentUser.setImageUrl(uri);
        Snackbar.make(findViewById(android.R.id.content),  text: "Image Uploaded", Snackbar.LENGTH_LONG)
                .setAction( text: "Action",  listener: null).show();
        //place image in profile image layout location
        Glide.with( activity: MenuActivity.this).load(uri).into(profile_image);

        //add the storage location uri to user db
        DocumentReference userRef = db.collection( collectionPath: "users").document(currentUser.getId());
        userRef.update( field: "profileUrl", uri.toString())
                .addOnSuccessListener(aVoid -> Log.d(TAG,  msg: "DocumentSnapshot successfully updated"))
                .addOnFailureListener(e -> Log.w(TAG,  msg: "Error updating document", e));
    })).addOnFailureListener(e -> {
        //if image upload fails, log error and display popup
        Log.d(TAG,  msg: "Profile image failed to update ", e);
        Snackbar.make(findViewById(android.R.id.content),  text: "Image not Uploaded", Snackbar.LENGTH_LONG)
                .setAction( text: "Action",  listener: null).show();
    }).addOnProgressListener(snapshot -> {
        //display uploading popup
        Snackbar.make(findViewById(android.R.id.content),  text: "Uploading", Snackbar.LENGTH_LONG)
                .setAction( text: "Action",  listener: null).show();


    });
}
```

Figure 41: Upload Image

This method uploads the image to Cloud Storage. This then returns the Uri for the image now uploaded. This Uri is then passed into the imageView using the “Glide” package. Once successfully written the new cloud Uri is uploaded to the user document to allow for the user image to be received when creating a new expense. Throughout this process, a snackbar appears to indicate the current upload status.

4.8.2 Dark Mode

```
//Enable/Disable darkMode
public void setDarkModeEnabled(boolean darkModeEnabled) {
    //set darkmode in user
    currentUser.setDarkMode(darkModeEnabled);
    //set app theme
    if (darkModeEnabled) {
        AppCompatDelegate.setDefaultNightMode(AppCompatDelegate.MODE_NIGHT_YES);
    } else {
        AppCompatDelegate.setDefaultNightMode(AppCompatDelegate.MODE_NIGHT_NO);
    }
    //update darkmode in db
    if (currentUser.getId() != null) {
        DocumentReference userRef = db.collection( collectionPath: "users").document(currentUser.getId());
        userRef.update( field: "darkMode", darkModeEnabled)
                .addOnSuccessListener(aVoid -> Log.d(TAG,  msg: "DocumentSnapshot successfully updated!"))
                .addOnFailureListener(e -> Log.w(TAG,  msg: "Error updating document", e));
    }
}
```

Figure 42: Set Dark Mode Enabled

The dark theme is set with a switch located in the User Menu. This switch activates the setDarkModeEnabled as indicated above. This method simply sets the default night mode to the user indicated and uploads the user preference to the database.

4.9 RECORD PAYMENT OUTSIDE APP

For the recording of payments received outside of the app, I simply added a received button to the payment object displayed on the incoming screen. The button when pressed updates the incoming document of that payment. It updates the fields; active: true, “payment_method”: Outside of App, and “date_paid” to the current system DateTime. These changes trigger the MarkasRecieved cloud function which then updates the respective due document of that payment.

4.10 SECURITY

4.10.1 Database Rules

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    allow read: if true;

    match /users/{userId}/{documents=**} {
      allow read: if true;
      allow write: if request.auth.uid == userId;
    }
  }
}
```

Figure 43: Security Rules

The rules for the Firestore Database are quite simple. They allow for any user to read from the database but only allow the users to write to the database of their document. When a payment is created in the incoming subcollection, this triggers a cloud function that creates the corresponding document in the beneficiaries document. The cloud functions on the server are run with administrator rights and can affect all documents freely.

4.10.2 Strong Customer Authentication (SCA)

“Starting September 14, 2019, new payments regulation is being rolled out in Europe, which mandates Strong Customer Authentication (SCA) for many online payments in the European Economic Area (EEA).” (Stripe, 2019) SCA requires two-factor authentication on payments. There are exceptions to this in which the additional authentication n is not required. IN most cases these exceptions are payments deemed as low risk. For Stripe Low risk is determined by a built-in risk analysis tool which among other things considers the amount to be paid. Most payment under the value of 30 EUR will be regarded as except unless otherwise flagged as risky by the bank. In this case, authentication will then be required. Which Stripe provides in the form of “3D Secure 2”.

4.10.3 3D Secure 2

This form of authentication is designed to provide a “frictionless authentication” method. In the instance of an android application, it is represented as a built-in authentication activity. For the testing mode, this application appears as Figure 44: 3D Secure 2 on the right. If the app were to go live, this would appear as a message indicating for the user to go to the banking application for the bank in which they are using. It may also request a one-time passcode which will be sent to the user's registered mobile number from their chosen bank.

“Other card-based payment methods such as Apple Pay or Google Pay already support payment flows with a built-in layer of authentication (biometric or password).” (Stripe, 2020) In testing Google Pay requires a biometric fingerprint input or password input in the payment overlay that appears when selecting Google Pay.

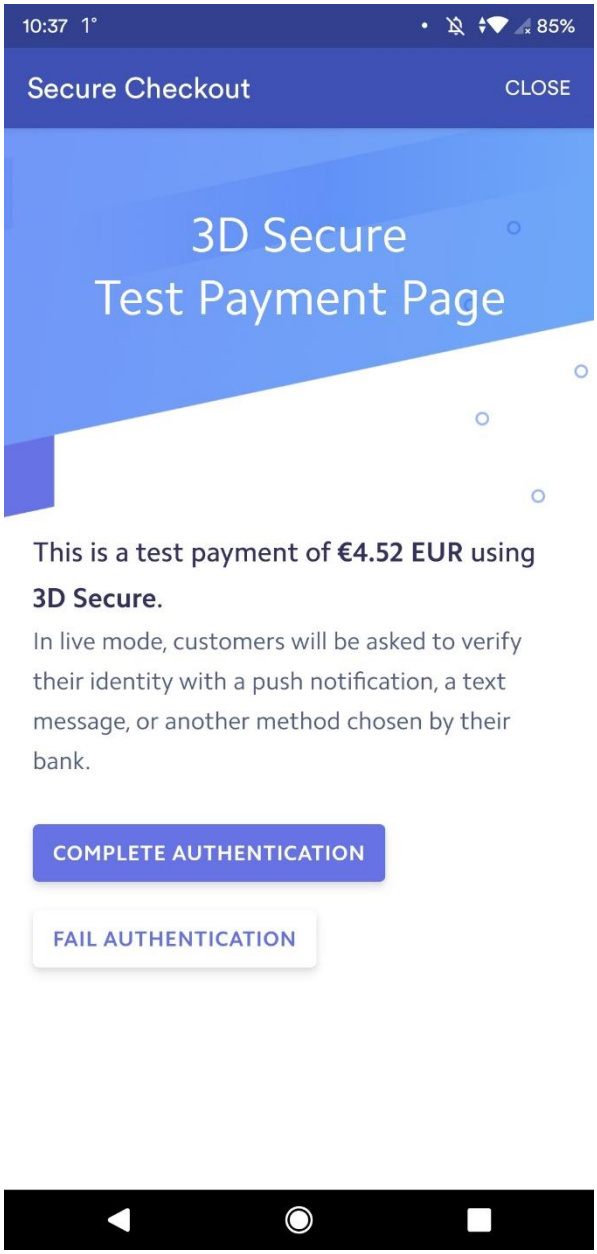

Figure 44: 3D Secure 2

5 TESTING

5.1 TESTING METHODS

Testing methods implemented for this application include:

Black Box Checking: Model test cases are written in this project, and manual testing is performed to verify the application's features.

White Box Testing: If the program satisfies user requirements and functionality according to test cases, the internal logic is thoroughly checked to ensure that there are no conceptual inconsistencies or problems in the application.

Integration Testing: After independently testing the features, they were tested by combining all the features in a single program.

End-to-End Testing: The entire application system is checked by linking the device to different machines, running it as an APK file, database, and local network. r

Usability Testing: Ultimately, usability testing is carried out by evaluating the flow of the application overall GUI design

5.2 TEST PHASE 1 – LOGIN AND REGISTRATION

For Login and registration, the testing implemented was Black Box. The test cases established were:

1. Test edit text fields for correct validation.
2. Email registration test.
3. Email login.
4. Google registration test.
5. Google login test
6. Facebook registration test
7. Facebook login test.
8. User deletion test.

Test cases 1, 2, 4 and 6 were completed without any issues. Tests 3, 5 and 7 produced an issue in which the user was required to complete login three times before the user becoming authenticated. I fixed this issue by adding an authentication listener. This provided enough time for the login to complete fully before checking for an authenticated user. Test 8, the user deletion required revision to add deletion of payments relating to

that user. Test 6 and 7; Facebook authentication later failed from an SDK update. This was fixed in Testing Phase 4.

5.3 TEST PHASE 2 – CREATE PAYMENT

For creating a payment, the test method used was primarily White Box with some Integration testing upon competition of logic checks for each feature.

1. Select/unselect users.
2. Split payment.
3. Create stripe payment for each user.

Test 1 produced varying success rates. This was resolved after creating an isSelected Boolean variable for each user that is created and attaching this variable to an indicator. Test 2 required multiple tests to ensure that the values indicated were correct. During these tests, I discovered that Stripe has a minimum payment value of 0.50 EUR. This limit was then added to the application. A checkbox was also added as a response to these tests for “Price includes yourself” which takes your portion of the expense cost into account when dividing the total price. Test 3 succeed without any issues.

5.4 TEST PHASE 3 – CHECKOUT

For the checkout, the testing implemented was Black Box. The test cases established were:

1. Pay with card (successful)
2. Pay with card (Authentication Required)
3. Google Pay

All test cases successful without any major modification.

5.5 TEST PHASE 4 – BUG FIXES

For this phase, I used End-to-End Testing and Usability Testing to run through the application and determine any hang-ups in performance or fix minor graphical and usability issues. From these tests, multiple minor issues were determined and quickly fixed.

It was in this phase that I also returned to the Facebook sign-in issue and repaired that by updating the SDK and update the hash keys of the devices authorised to produce a certified APK of the application. It was in this phase that I also cleaned up the code and

added my code comments. During this clean up I added the currentUser object to reduce the number of document calls required and overall improve performance significantly.

5.6 REMAINING ISSUES.

Any issues which did affect performance, usability or functionality significantly were removed and replaced with alternate methods. Some issues remain at the time of FYP completion. These issues are issues that do not break the overall functionality of the application.

5.6.1 Facebook Sign Out

When signing out from Facebook. The application appears to freeze. This is caused by a significant delay in signing out when using a Facebook account. This issue does not present itself at every instance but is only present with a Facebook-based auth. The cause of this is unknown but as the application does not report an error it appears that this is a limitation to the Facebook SKD and Firebase integration. This issue is only significant in testing as in day-to-day use, users will not be logging into multiple accounts.

5.6.2 .Cloud Function Deployment

The cloud functions are normally deployed from the firebase CLI on the development pc. An error occurred which prevented the function to be deployed fully from said CLI. The solution to this was to:

1. Deploy function from the CLI. This fails to deploy but successfully uploads the function to the cloud functions server.
2. Direct admin to the online cloud function CLI, located at https://console.cloud.google.com/functions/list?project=pay-up-66b7c.
3. Click into the selected function.
4. Select “Edit”
5. Deploy the function without making any edits to the function.

The function will then successfully deploy from the online CLI. The entire process of creating a function can be implemented from the CLI but the method of doing so is cumbersome and overall slower than the method mentioned above.

As this is only an issue when modifying the functions, does not apply a significant delay and does not impact the user functionalities., this issue was deemed minor.

6 GRAPHICAL USER INTERFACE

6.1 SIGN IN/ REGISTER

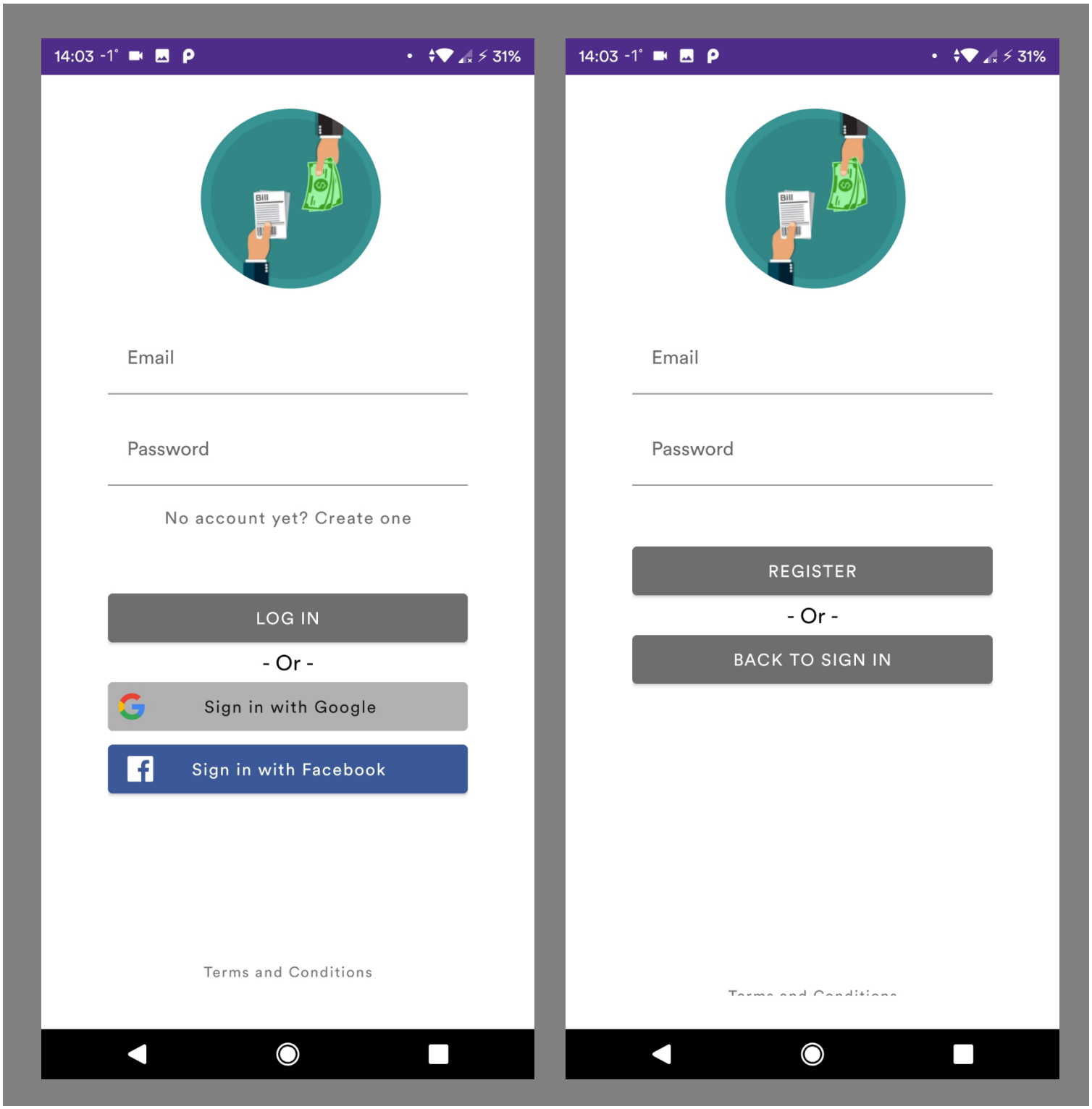

Figure 45: Sign In / Register GUI

6.2 PROFILE SETTINGS

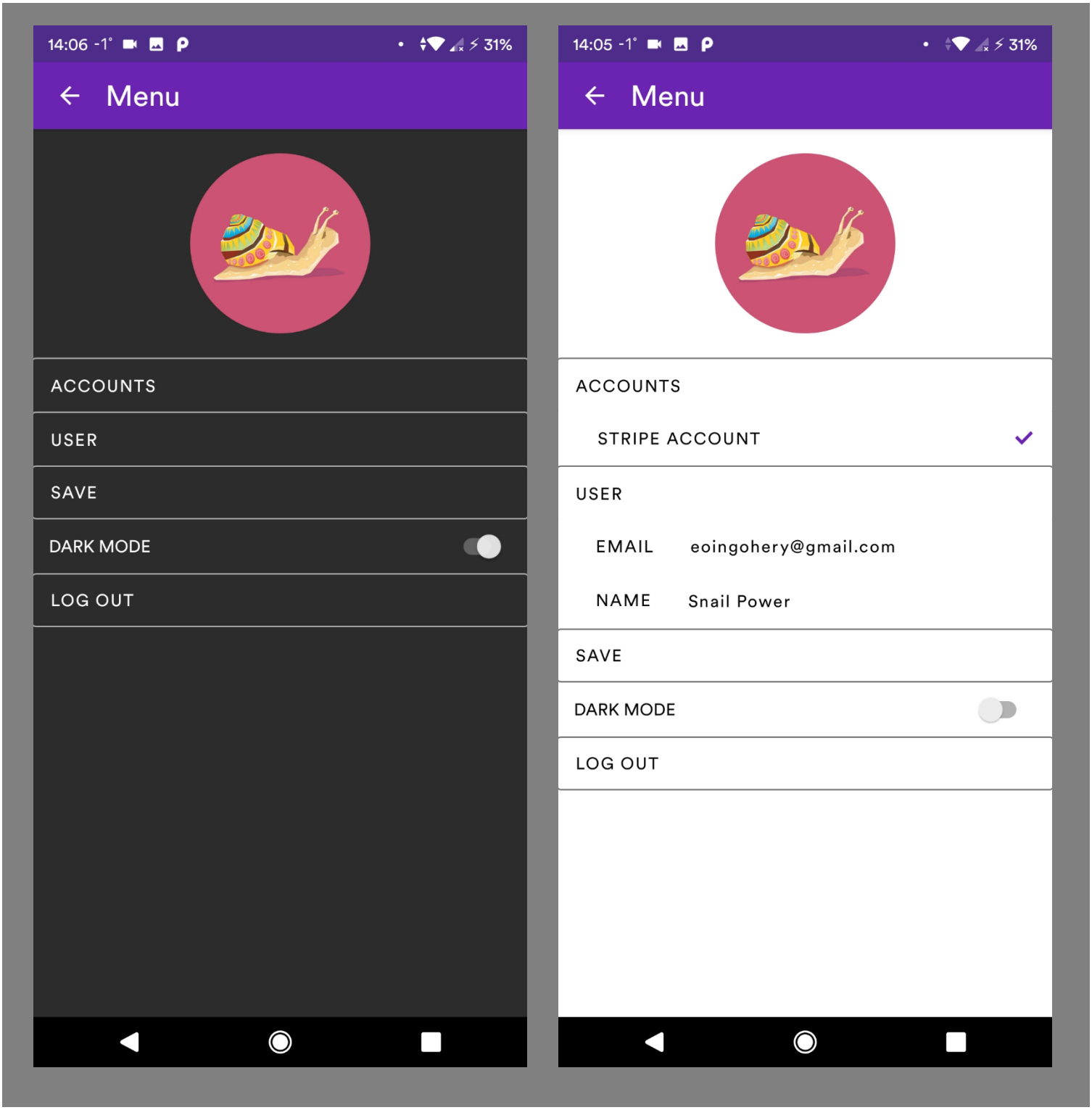

Figure 46: User Menu GUI

6.3 DUE FRAGMENT

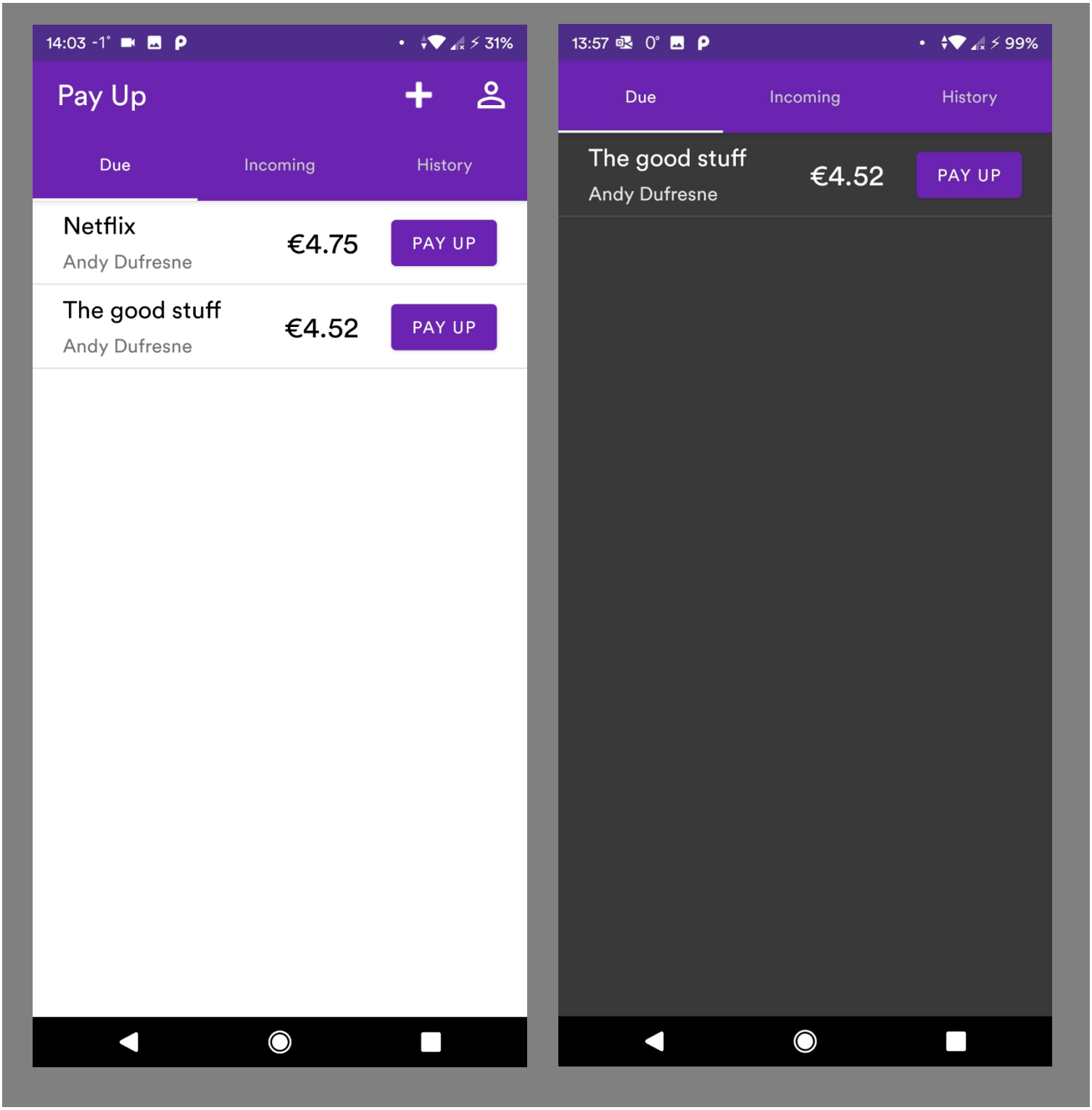

Figure 47:Due Fragment GUI

6.4 INCOMING FRAGMENT

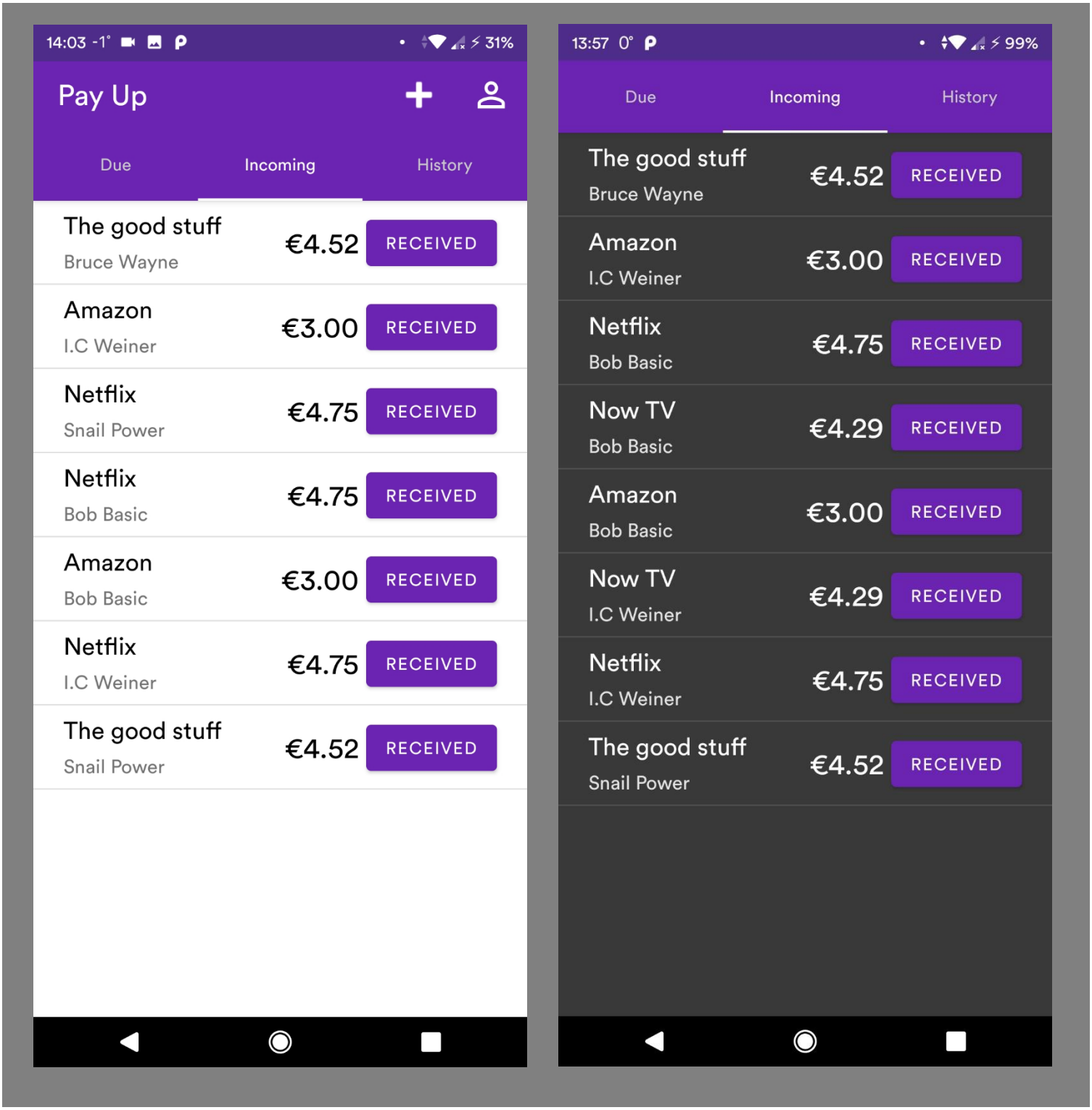

Figure 48:Incoming Fragment GUI

6.5 HISTORY FRAGMENT

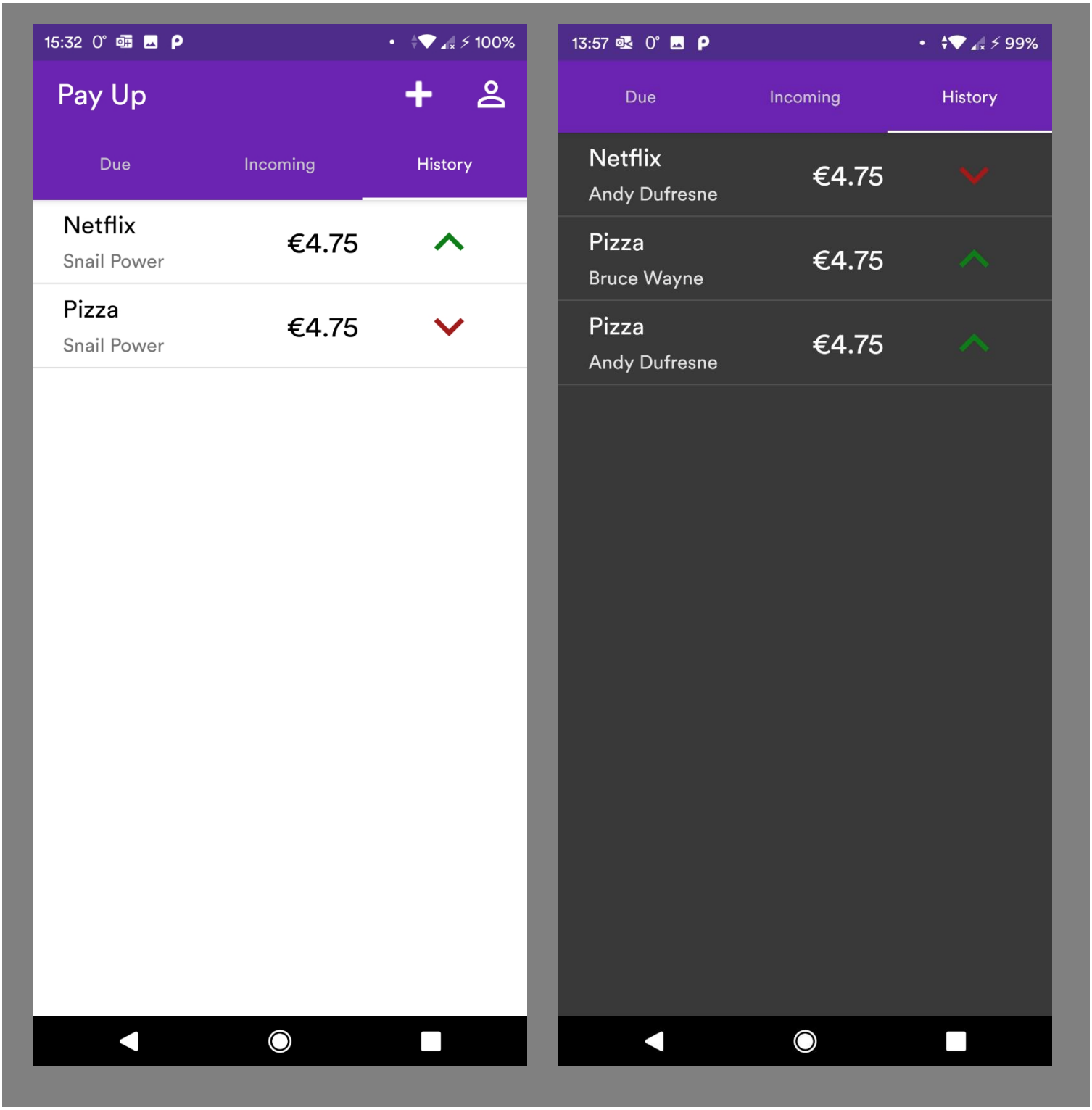

Figure 49: History Fragment GUI

6.6 PAYMENT DETAILS

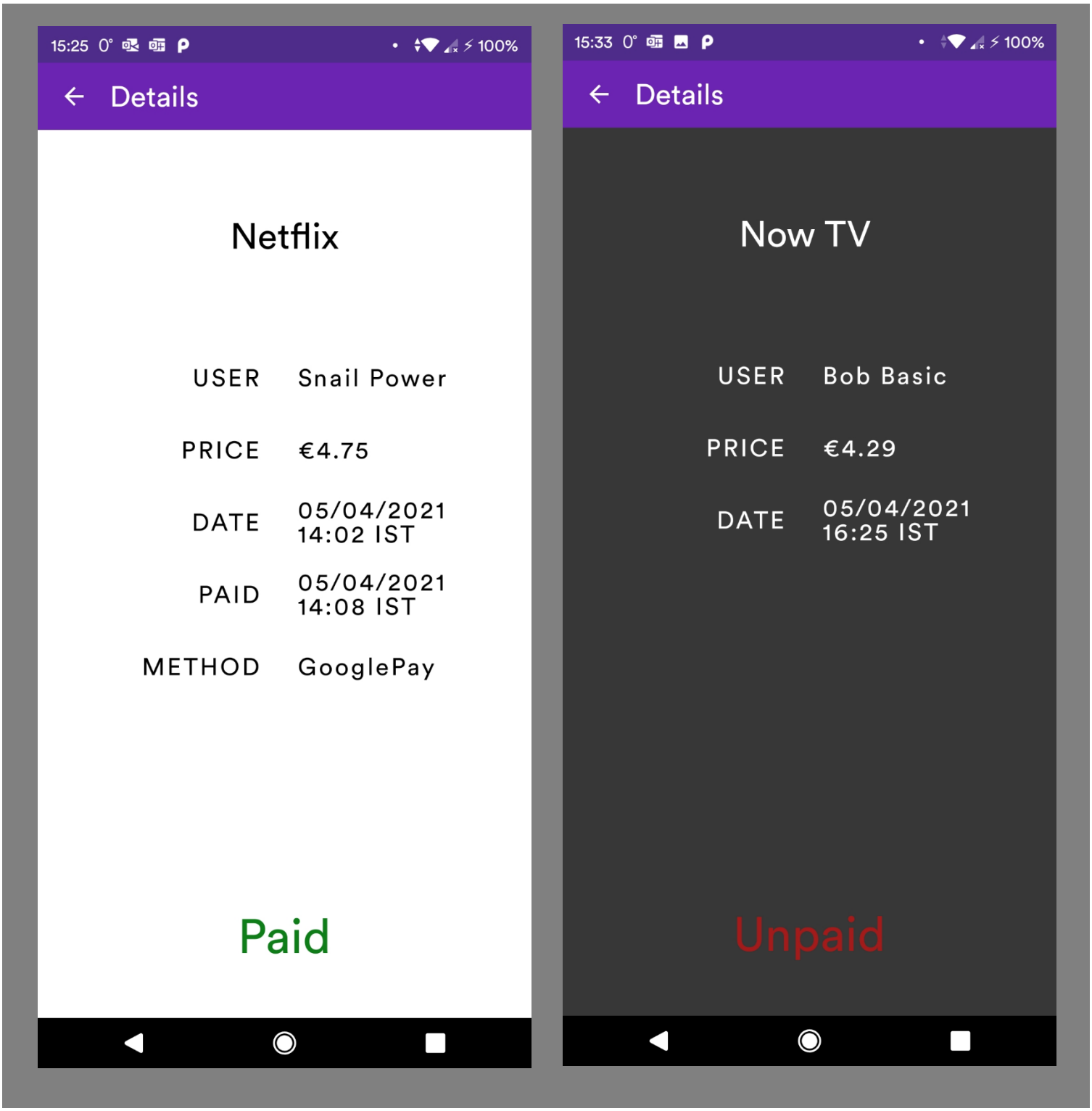

Figure 50:Payment Details GUI

6.7 CREATE AN EXPENSE

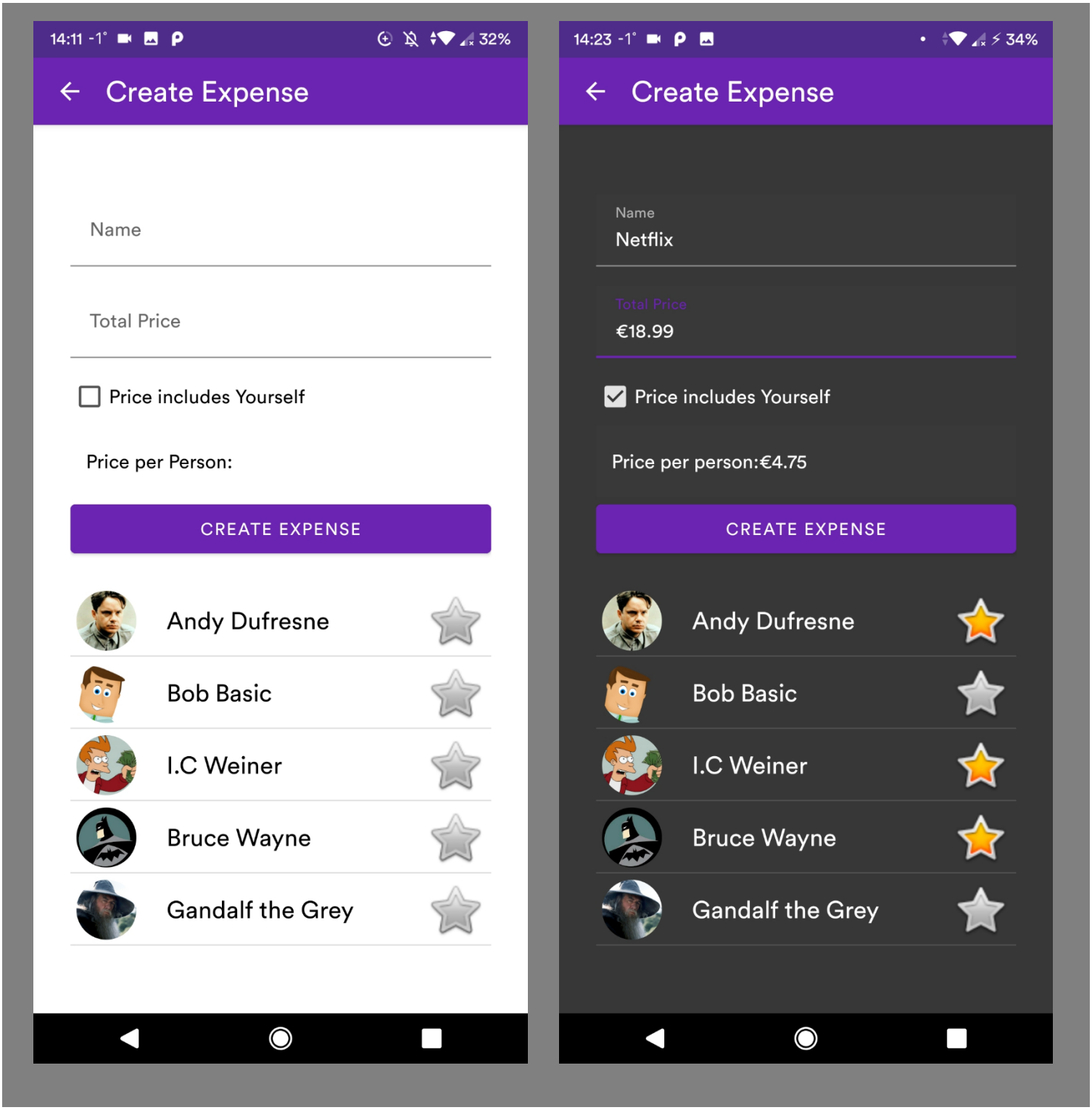

Figure 51:Create an Expense GUI

6.8 CHECKOUT

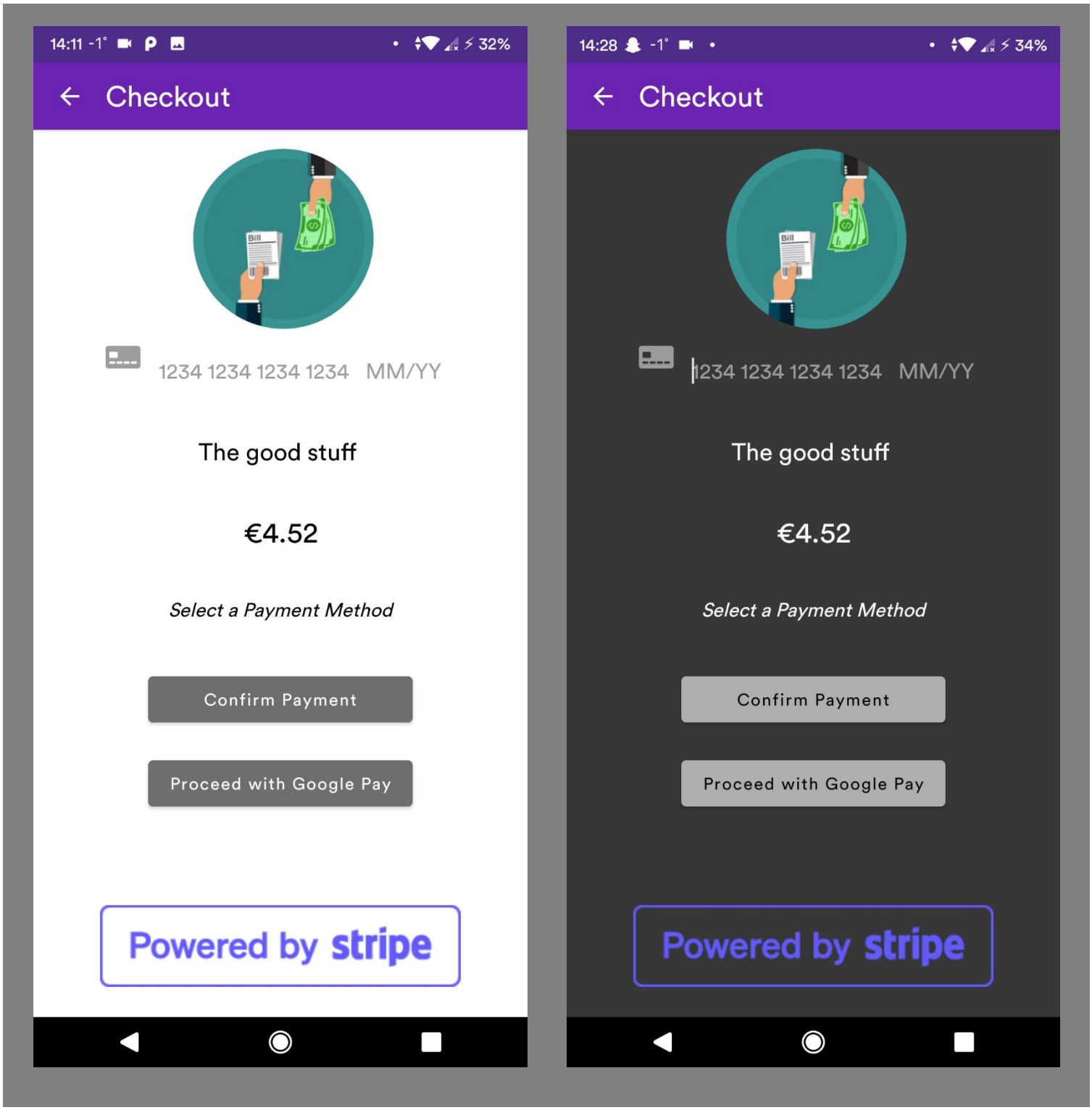

Figure 52:Checkout GUI

6.9 STRIPE ONBOARDING

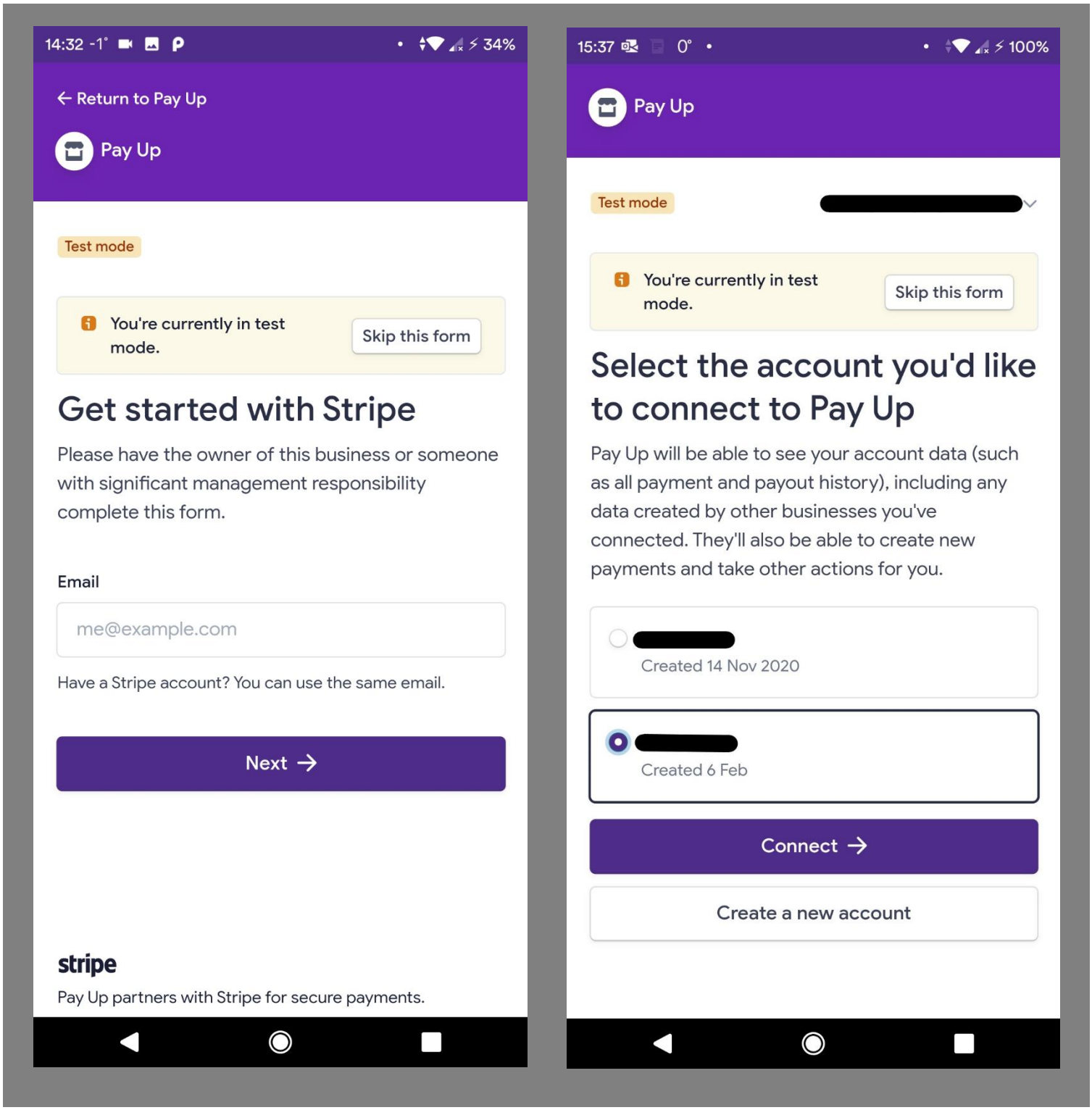

Figure 53: Stripe Onboarding

7 CONCLUSIONS

7.1 ORIGINAL OBJECTIVES

For this application to be a success, I must evaluate my results from the perspective of the objects that I set out to achieve when I began this project.

7.1.1 Research Current Digital Payment Methods

After engaging in the project my understanding of digital payments and the security involved improved significantly. From the initial research of the different method of payments and their level of usage in society to the complexities required to facilitate online payments, I feel that my understanding of these concepts is far more defined hen previously. I now understand better the concept of SCA and why it is considered essential in the security of digital payment.

7.1.2 Design a System

As indicated by the existence of this documentation, I believe I was successful in designing a system capable of implementing the features allayed out under Requirements. The use cases of the application were established, and the overall architecture of the system was designed to be applied in a functioning prototype.

7.1.3 Implement Prototype

I have created a working prototype of this application as detailed in the Implementation section of this document. Although the application has not gone. I believe that the available APK located on GitHub demonstrates the functionality of the app and proves that my attempts to create a working prototype were successful.

8 FURTHER DEVELOPMENT

8.1 RECURRING PAYMENTS

The base Stripe API does offer the ability to create a recurring payment, but this feature is not easily accessible when using Stripe connect. This is due to the inability to create recurring payment intents that allow for the depositing of funds directly to a connected account. The current implementation of recurring payment is designed for the sale of company services and as such the company using stripe (in this instance, me) is to receive the payments directly.

This can be worked around but requires a far more intensive usage of Stripes services and would not be possible with the current cloud functions server as the google cloud platforms limit HTTP request packets to 10MB. All automated stripe HTTP calls, such as the success of intent payment object confirmation, exceed this limit and would be required to effectively implement recurring payments

8.2 SAVE PAYMENT DETAILS.

The ability to save payment details can be implemented. The payment details are saved to Stripe to be retrieved later if required. It requires the use of two different checkout methodologies combined. As it is not essential to the functionality of the application and the application is only designed to be a proof of concept, I chose to leave this feature out.

I have left in functions and classes that would facilitate this in the future as it was part of my original method for confirming stripe payments in the checkout. I chose to move away from this method as it did not allow for Google Pay.

8.3 PAYPAL

Currently, the method of integrating PayPal into an android application could be done through the PayPal Android SDK: https://github.com/paypal/PayPal-Android-SDK. “The SDK will now use the newest version of the PayPal Wallet App if present on the device to log in to a customer account.” (PayPal, 2019) This SDK will soon be deprecated and as such PayPal have requested any new application to use the Braintree service. The Braintree service functions similarly to stripe. It acts as a payment gateway capable of accepting multiple payment methods including digital wallets but requires a far more comprehensive server to run the required functions with larger bandwidth for HTTP requests and has less detailed developer documentation. A potential future version of this application could see Braintree implemented as an alternative to Stripe.

9 REFERENCES

Edelman Intelligence. (2020, 06 25). *TRUST BAROMETER SPECIAL REPORT: BRAND TRUST IN 2020*. Retrieved from Edelman: https://www.edelman.com/research/brand-trust-2020

App Annie. (2020). *State of Mobile 2020.* App Annie.

Epstein, A. (2016, 03 14). *A third of Netflix watchers in the US don't pay for Netflix*. Retrieved from Quartz: https://qz.com/638598/a-third-of-netflix-watchers-in-the-us-dont-pay-for-netflix/

Google. (n.d.). *Firebase Documentation*. Retrieved 12 02, 2020, from Firebase: https://firebase.google.com/docs/auth/admin/create-custom-tokens#:~:text=Firebase%20gives%20you%20complete%20control,JSON%20Web%20Tokens%20(JWTs).&text=Furthermore%2C%20the%20contents%20of%20the,Security%20Rules%20and%20the%20request.

MindSea Team. (2020). *28 Mobile App Statistics To Know In 2020*. Retrieved from MindSea: https://mindsea.com/app-stats/

Moeser, M. (2020, 07 22). *The precarious rise of subscription payments*. Retrieved from Payments Source: https://www.paymentssource.com/list/the-precarious-rise-of-subscription-payments

Nawaz, Z. &. (2017). Simplified FDD Process Model. *International Journal of Modern Education and Computer Science*, 53-59.

PayPal. (2019, 01 15). *PayPal-Android-SDK*. Retrieved from GitHub: https://github.com/paypal/PayPal-Android-SDK

Salmony, M. (2017). The future of instant payments: Are we investing billions just for mobile peer-to-peer payment? *Journal of Payments Strategy & Systems*, 58-77.

Shaw, N. (2014). The mediating influence of trust in the adoption of the mobile wallet,. *Journal of Retailing and Consumer Services*, 449-459.

Shin, D.-H. (2009). Towards an understanding of the consumer acceptance of mobile wallet. *Computers in Human Behavior*, 1343-1354.

Stripe. (2019, 07 18). *3D Secure 2*. Retrieved from Stripe.com: https://stripe.com/en-ie/guides/3d-secure-2

Stripe. (2019, 09 14). *Designing payment flows for SCA*. Retrieved from Stripe.com: https://stripe.com/en-ie/guides/sca-payment-flows

Stripe. (2020, 07 31). *A guide to payment methods*. Retrieved from Stripe: https://stripe.com/en-ie/payments/payment-methods-guide

Stripe. (2020). *Security at Stripe*. Retrieved from Stripe Docs: https://stripe.com/docs/security/stripe

Stripe. (2020, 05 04). *Strong Customer Authentication*. Retrieved from Stripe: https://stripe.com/en-gb/guides/strong-customer-authentication

Stripe. (n.d.). *Using Android basic integration*. Retrieved 2020, from StripeDocs: https://stripe.com/docs/mobile/android/basic

Tony Chen, K. F. (2018, 02 09). *Thinking inside the subscription box: New research on e-commerce consumers*. Retrieved from McKinsey & Company: https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/thinking-inside-the-subscription-box-new-research-on-ecommerce-consumers#

UK Finance. (2020, 06). *UK Payment Markets 2020*. Retrieved from UK Finance: https://www.ukfinance.org.uk/system/files/UK-Payment-Markets-Report-2020-SUMMARY.pdf

Worldpay Editorial Team. (2019, 08 06). *Comparing eWallets Apple Pay, Samsung Pay, Google Pay*. Retrieved from Fis Global: https://www.fisglobal.com/en/insights/merchant-solutions-worldpay/article/apple-pay-vs-samsung-pay-vs-android-pay

Wurmser, Y. (2020, 07 09). *The Majority of Americans' Mobile Time Spent Takes Place in Apps*. Retrieved from eMarketer: https://www.emarketer.com/content/the-majority-of-americans-mobile-time-spent-takes-place-in-apps

Zoran Kalinic, V. M.-C. (2019). A multi-analytical approach to peer-to-peer mobile payment acceptance prediction. *Journal of Retailing and Consumer Services*, 143-153.

10 APPENDIX

10.1 GITHUB

GitHub Url: https://github.com/EoinGohery/PayUp

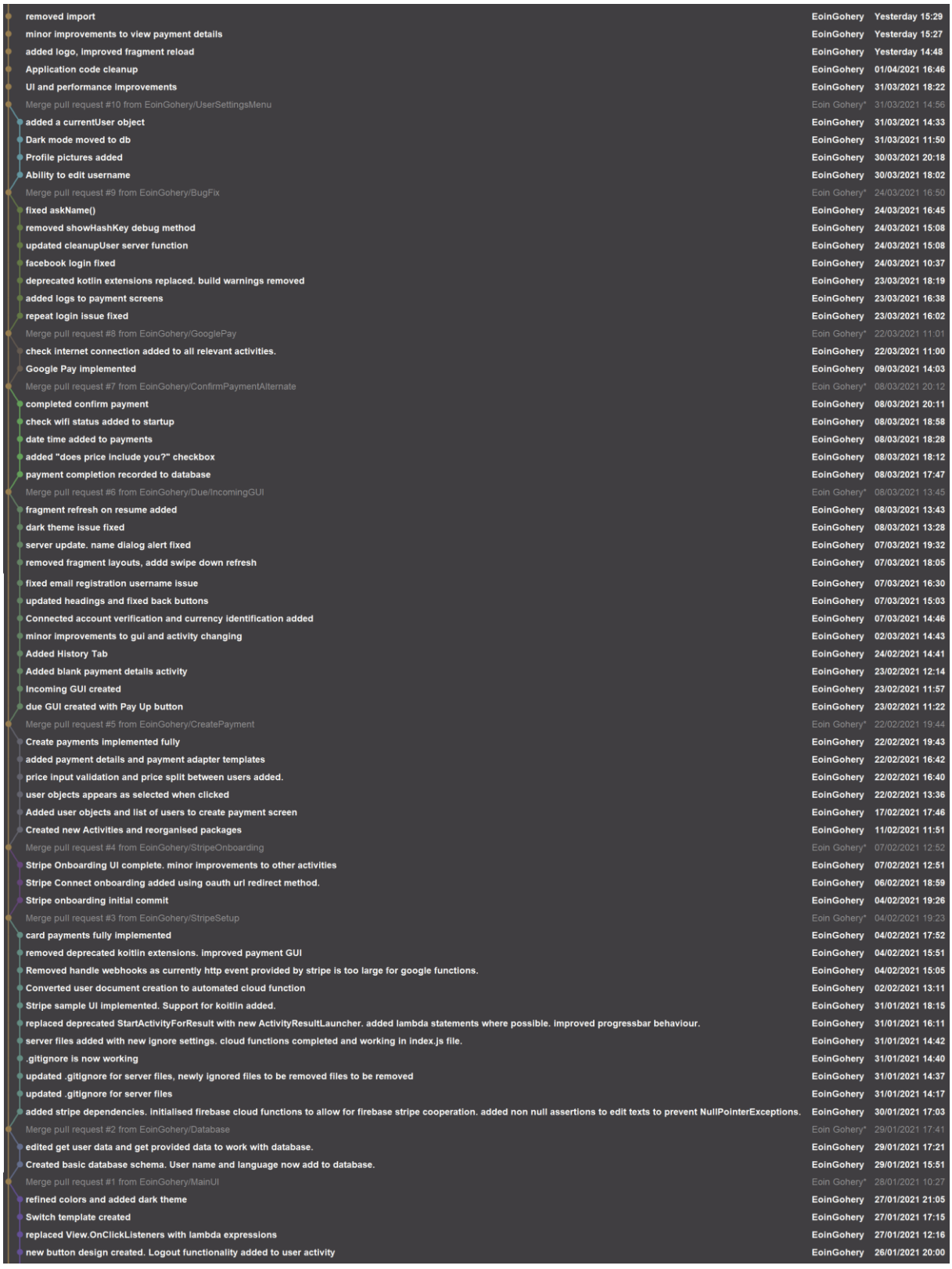

Figure 54: GitHub Log

10.2 JOURNALS

10.2.1 October 2020

10.2.1.1 Monday 12th

This was day 1. I set up the template for my FYP report. This involved formatting the document and determining the areas in project development that require detailed descriptions. I also put together a rough project plan which I will likely change further, and hopefully define fully by the end of this month or early November after initial research and project meetings with my supervisor. These meetings will help to determine key dates and project goals and will most likely affect the overall flow of the project. The idea of the project as it stands now is likely to change and may continue to change throughout the entirety of the project. As such the project plan which you see in this document will only represent a rough description of the project and not an exact process breakdown.

10.2.1.2 Thursday 15th

This was the first day in which the project moved beyond a pure research phase. I set up the initial project on Android studio using Lollipop 5.1. I chose this version of Android for the high compatibility of the version. If needed for the project to develop beyond a major block, the existing code can be moved to a different version of Android.

10.2.1.3 Friday 23rd

GitHub repository for the project is set up and an initial commit is made. The tool LingoHub was implemented into the project. This was done by implementing setting up the LingoHub pro account, which was provided free as a GitHub Student member. The location of the strings.xml file was provided to LingoHub. With this setup, LingoHub will now activate upon new commits. It will check for changes to the strings.xml file and translate the strings to the selected languages and commit the changes to GitHub. This provides a simple and relatively accurate method to translate the app to multiple languages. For the initial development of the project, I chose English, French, Chinese and Arabic as the initial Application languages available. This will be added to in later stages of the process as there is no limit to how many languages can realistically be implemented into the project.

10.2.1.4 Tuesday 27th

This day was spent determining the base requirements for the application. This included the functional requirements primarily. Some minor user and non-functional requirements

were also defined at this time but have not been fully developed upon at the moment. I also added additional headings to my report and re-evaluated others.

10.2.2 November 2020

10.2.2.1 Saturday 14th

Changes were made to the base idea and as a result, a new GitHub repository needed to be created. I created this new GitHub under the name “Pay Up”. I also set up the firebase account for this project and linked the app to this account. I then began setting up a basic layout frame using the new “layout” based system which has been added to Android Studio since my last android application. This new system is used as a replacement for the old “constraint” based layout system. This new system took time to adjust to, but a basic frame has now been set up in this method for the login and main homepage

10.2.2.2 Sunday 15th

Login Gui is implemented and adjusted to near its final version after multiple iterations throughout the day. Google login and register are also implemented and tested with multiple google accounts to ensure that the Authentication database was updating correctly.

10.2.2.3 Monday 16th

Email login is implemented with email and password inputs. Valid email and password checker implemented with an indicator in test box on validation fail if a field is blank. Authentication popup implemented to prevent user input while login is underway. Registration to be implemented at a later date

10.2.3 December 2020

10.2.3.1 Thursday 3rd

Reviewed research of the previous idea and adopted it to account for the new idea. Research on android and digital payments can be adapted to the new concept. All other research is no longer applicable.

10.2.3.2 Monday 7th

Completed research on subscription services and the culture of sharing accounts. Research consolidated and reviewed for documentation.

10.2.3.3 Friday 11th

Research for payments gateways completed. Research compiled and added into report.

10.2.3.4 Wednesday 16th

Research completed for mobile wallets and other payment methods. Research also completed for security. Documentation updated

10.2.3.5 Sunday 20th

Functional and environmental requirements adapted from previous idea and detailed ion report.

10.2.3.6 Tuesday 29th

Non-functional and User requirements adapted from previous idea and detailed ion report.

10.2.3.7 Thursday 31st

Use cases for Login and Registration created with support for email, Google, and Facebook. The project plan for the previous idea converted for the new idea

10.2.4 January 2021

10.2.4.1 Saturday 2nd

The registration form layout is created with basic validation to check for blank fields. Account registration authentication is implemented but currently not working correctly.

10.2.4.2 Sunday 3rd

Authentication for email and password registration and login is fixed. The problem was simply that I was passing the Edit Text to Firestore and not the value in the input field.

10.2.4.3 Tuesday 5th

Facebook developers account is created and a “Pay Up” project is created. The application is then linked to this account to allow for the Facebook API to be utilised. For this application, the only feature that will be implemented is the Facebook login. Facebook login and registration implemented. Get user and provider data added.

10.2.4.4 Thursday 7th

Time spent on “Introduction” within documentation with further research added and references updated.

10.2.4.5 Tuesday 12th

Began testing on Login and registration by all available methods.

10.2.4.6 Friday 15th

Completed testing on login, testing documentation to be completed and added to report.

10.2.5 February 2021

10.2.5.1 Wednesday 3rd

Added Handle Event Webhook configuration to the server-side and connecting this cloud function to the stripe server.

10.2.5.2 Thursday 4th

Removed handle webhooks events due to HTTP request size limit on cloud functions. An alternate server would be required to run the webhooks but the webhooks are not a necessity anyway. Completed implementation for the basic card payments. Started the onboarding process

10.2.5.3 Saturday 6th

Added stripe onboarding by implementing the pre-built OAuth URL. The app simply redirects the user to this website with an embedded browser. When the user is returned the user account authentication code is provided by the redirect URL. Cloud function implemented to handle the authentication process.

10.2.5.4 Sunday 7th

Minor changes to improve stripe onboarding. Most changes made to the backend cloud function to ensure better error logging. Merged onboarding into the main branch.

10.2.5.5 Thursday 11th

Today I just sorted the project structure around. Broke my activities into packages. I then did some minor visual improvements to the fragments and added the create payment activity for later use. I will attempt to create this activity in Koitlin.

10.2.5.6 Wednesday 17th

Started the create payment activity by creating the required user object to appear in a recycler adapter. This will allow the user to select which users they would like to add to the payment. Users that are added are moved from all recycler to the added Users recyclers.

10.2.5.7 Monday 22nd

Changed the user selection method from two separate recyclers to a single recycler. The user now has a star next to it which becomes highlighted on clicked the user. Added field input validation. Price now splits between the selected users. Added the payment objects are adapter for said objects. These functions similarly to the user object. Create Payment is merged into master.

10.2.5.8 Tuesday 23rd

Updated Gui for the payment objects to appear in the adapters. GUI for the fragments updated. Added a blank activity for viewing the payment details.

10.2.5.9 Wednesday 24th

Added a history tab to view payments that have been paid. These expenses will no longer appear in their respected fragments (due, incoming)

10.2.6 March 2021

10.2.6.1 Tuesday 2nd

Just made a few minor GUI improvements and tried to improve the performance when changing activities.

10.2.6.2 Sunday 7th

Added a method to verify if a user has set up a connected account on stripe. This will appear as a tick or x next to “Stripe Account” in the user menu. Fixed some of the headings and basic buttons. I tried to change the way the names are added to the database on registration to allow for users to set their desired username. This is not working yet. Added a swipe down to refresh on the fragments.

10.2.6.3 Monday 8th

Fixed a dark theme issue. I was extending Activity and not ActivityCompat. As a result, my rooted phone was able to overwrite the standard dark theme settings on only the payment details activity. Finished the card payments using a confirmed payment functionality of the Stripe API within the application (no longer part of the server functions). Database updates to record the completion of the payment.

10.2.6.4 Tuesday 9th

Google pay is implemented and tested. So far it appears to be working.

10.2.6.5 Monday 22nd

Added a check internet connection method to ensure that the app cannot be accessed without a working internet connection. Merged the google pay to master.

10.2.6.6 Tuesday 23rd

Finally fixed the login issue that required the user to log in 3 times before successfully redirecting to the main activity. I did this by replacing the old check current user with a new auth listener. Now when a change in the authentication status is detected the user is redirected to the main. Removed the deprecated Koitlin extension.

10.2.6.7 Wednesday 24th

Fixed the Facebook login that was broken by an update to the Facebook SDK. updated cleanUpUser cloud function. Now when a user is removed, all of the payments relating to them are removed from the database. If a user is removed any payments relating to them are removed. Fixed the askname to only ask for names upon initial registration.

10.2.6.8 Tuesday 30th

Added the ability for the user to upload a profile picture that appears in the create payment activity. Added an edit text to the user menu so that the users can now edit their names at any point.

10.2.6.9 Wednesday 31st

Removed the shared preferences in favour of storing all user settings in the database. Added a current user object that extends from the original user object. This allows for all data of the current user to be stored in a publicly available object. Improves performance by removing the need to constantly refer back to the database.

10.2.7 April 2021

10.2.7.1 Thursday 1st

Cleaned up the code, reorganising and adding comments where necessary.

10.2.7.2 Monday 5th

Completed the Demo Day video. I went for a short marketing style video that runs through all the front-end features of the application.

10.2.7.3 Tuesday 6th

Fixed the onboarding. During the code clean-up, I removed the get user profile from the result hander of the stripe onboarding.