Finances Management App

Rafael Gómez Pérez

Resum— Este proyecto intenta cubrir las necesidades que todo grupo de amigos se encuentra y, suele ser el centro de la mayoría de problemas, el dinero. Nos ha pasado a todos que salimos a hacer algo con los amigos y, a la hora de pagar, alguien no tiene suficiente dinero por lo que otro miembro del grupo paga su parte. Es fácil que, con el paso del tiempo, nos olvidemos de quien le debe dinero a quien, el motivo y la cantidad, y esto puede resultar en peleas y/o enfados. Ya sea que la persona que debe dinero carece de la moral para intentar saldar su deuda lo antes posible, o deja pasar el tiempo sin comentarlo para que el tema se olvide, es díficil acordarse de todo esto. Para resolver ese problema hemos creado la app llamada WhoPays. WhoPays permite al usuario, de manera rápida y sencilla, mantener un historial claro de todos los eventos que ha realizado ese grupo y cuanto dinero debe cada miembro de éste.

Pataules clau—Programación Reactiva, Arquitectura de Microservicios, I/O No bloqueante, Spring Webflux, Aplicaciones Híbridas Ionic

Abstract—This project tries to cover the need that every group of friends has and is always the center of problems, money. It's happened to everyone that we go out with friends and someone doesn't has the money to pay and someone has to pay for him. It's easy as time goes by to lose track of that money, whether the guy who owns money lacks the moral conviction to pay it as soon as possible and tries to not talk about it in order for everyone to forget it or the guy who paid simply doesn't think there's a problem there. In order to solve that problem we've created the app called WhoPays. An easy to use app that allows the users to track every member of his multiple groups debts with ease and hold them accountable.

Index Terms— Reactive Programming, Microservices Architecture, Non-Blocking I/O, Spring Webflux, Ionic Hybrid Apps

1 Introduction

The concept of finances management mobile applications is not something new. There are a couple of mobile applications in the market currently that provide it's users with similar functionalities.

This project's goal is to develop a fully functional application that allows us to easily manage our expenses within a friend group. In order to develop the project, we've chosen to implement a Microservice architecture (MSA) with Spring Webflux on the backend and Ionic 4 **[1]** to create the hybrid mobile application on the frontend.

The most common topic that creates disputes amongst friends is money. Everyone has friends in some of his groups that are always low on money and need to be paid for meals that will never be paid back.

In order to solve and prevent that situation we created WhoPays. WhoPays is a functional mobile application that allows us to keep track of what expenses each and every group we are in has so we can see who owns money to who.

Even though the most visible part of the project is the mobile application because that's what the user sees, the

development's team focus has always been creating a proper and well designed implementation of a microservice architecture.

2 State of the art

Developing this mobile-based application is not something completely new. There are some other applications in the market that provide similar functionalities however, we wanted to be able to create a similar solution that could be later be further customized for our own needs.

We see the other applications in the market as an all-rounder whose target is a large amount of people while our application's target is not to be used by millions of people but our own group of friends so that we can create new features that could be good for our needs but not necessarily to every user.

One clear example of that is Tricount **[2]**. Tricount is the most used finances management application that allows it's users to track the expenses of the members of their groups.

Among all the functionalities Tricount provides expenses tracking, user's debt and a visual representation through charts of how much is each user's debt. But one of the problem that we thought Tricount had was that, unlike other group base apps like WhatsApp there is no administrator of the group.

Without any admins nor role privileges every member can add or change any expense data like reduce his debt or increase someone's.

In order to prevent that WhoPays added the admin functionality which only allows users categorized the admin role to create or update any current or previous expenses of the group.

Another difference that we found after using Tricount for some time is the fact that when we create an expense in Tricount, we have to fill a form. In web based applications forms are a standar input for user data but in mobile based applications forms are somewhat tedious and you eventually grow tired of having to fill a whole form when you want to create a simple expense.

What we came up with to solve that problem was to make use of one of Ionic's native **[3]** mobile functionalities, Speech Recognition **[4]**. Speech Recognition is a feature provided by the Spring Native module that allows us to transform any audio input source into text.

By using speech recognition we were able to allow our users to fill most of the tedious form by simply clicking a button and speaking to the phone. We want to make clear that not all fields can be filled with speech recognition, for example the list of guys who own money or who they own it to is not something that speech recognition allows us to fill.

3 PROJECT PROPOSAL

WhoPays is an application that solves the problem of financial disputed among a group of friends. What the app provides is a simple way to identify which group members from the group owns money (quantity and who is owned).

Through simple charts we can recognize almost immediately who are the members that own money, so we can hold them accountable and make them pay at the next event or tell them to make us a transfer into our bank account.

After looking at similar projects and having multiple conversations with possible stakeholders this are the key requirements that we got:

- **Req-1:** The app must have an easy way to show which members own money
- **Req-2:** The app must have a history of the events/expenses of that group
- **Req-3:** The app must have a list of all the members of the group
- **Req-4:** The app must have an easy way to introduce new expenses (not forms)

To facilitate the fact, if necessary, of applying changes in the future the WhoPays app will be needed to be as scalable as possible. That along with speed and fault tolerance made us go with a microservices architecture **(MSA)**.

3.1 Project Objectives

From the main requirements explained at the previous section, we can extract some more specific objectives that will be the conductive thread milestones that will be carried out during this project.

-**TFG-OBJ-01** – Usage of the Software Development knowledge acquired throughout my specialization. Consists of implementing the knowledge I've acquitted to the whole Software Development Lifecycle (Planning, Analysis, Design, Implementation, Test, Document).

-**TFG-OBJ-02** – Polish my Software Development skills. Consists of further my knowledge on every aspect of the Software Lifecycle in order to become a better professional.

-**TFG-OBJ-03** – Learn about the best tools in Software Development. Consists of learning new frameworks (Spring Webflux), methodologies (Kanban), tools (Docker**[5]**) and architectures (MSA) that are becoming the standards of our field in order to get used to using them everyday.

-**TFG-OBJ-04** – Usage of Clean code and Design Patterns. Consists of using the standards of Clean Code throughout the whole project (according to the Clean Code book), and getting familiar with a couple of the most commonly used Design Patterns like Visitor or Factory Method.

-**TFG-OBJ-05** – Create an application to track expenses with ease.

-**TFG-OBJ-05.1** – Track and manage the expenses of different groups of friends within the same application with an easy to understand UI.

-**TFG-0BJ-05.2** – Track and manage the home expenses of the user with an easy to understand UI.

-**TFG-OBJ-06** – Create a microservice architecture and clustering. Consists of implementing an **MSA** with the help of cloud clustering and deployment to speed performance and have fault tolerance.

-**TFG-OBJ-07** – Create a multiplatform application. Consists of creating a hibryd application with Ionic so that can be deployed in both mobile OS (Android and iOS).

4 WORKING METHODOLOGY

To develop WhoPays we thought of using a methodology that could allows us to develop features in an agile way while being able to focus on finishing a specific feature before moving onto the next one.

However, we didn't find any existing methodology that would fit with developing centered around features while keeping a visual representation of the workflow, but we found a way to merge two already existing methodologies into one that would fit our requirements.

4.1 Working Methodologies: Kanban

Kanban is a mean to design, manage and improve the flow of work. It provides a visual representation of the flow of work where we can see the state of our issues (PLANNED, IN PROGRESS, TESTING, DONE, RELEASED).
Kanban really excells in an environment where we want to deploy work as soon as it's ready because we will know the exact moment a feature has been finished.

Among the multiple benefits of using Kanban this are the ones that usuaylly are the easiest to recognize:

- **Transparency** : sharing information openly using clear language improves the flow of business balue.
- **Balance** : different aspects, viewpoints and capabilities must be balanced in order to achieve effectiveness.
- **Flow** : Work is a continous or episodic flow of value.
- **Understanding** : Individual and organizational self-knowledge of the starting point is necessary to continue improving.

In order to use Kanban to it's fullest potential there's a list of practices that we have to follow within a Kanban system:

- **Visualize** : use a kanban board in order to show the team WIP limits, state of all issues and the delivery point to a client. In our case we had a github Kanban board at every module of our project (Front end and Backend) so we could have a visual representation of the work.
- **Limit work in progress** : limit the amount of work you have in progress in a system and use those limits to guide when to start new features. In our project we adjusted the WIP by multiplying the number of developers twice (2 x 1) so we could only have 2 tasks in develop stage at the same time.
- **Manage flow** : try to maximize the delivery value while minimizing lead times and be as predictable as possible. A key aspect of flow management is identifying and addressing bottlenecks and blockers.
- **Implement feedback loops** : feedback loops are an essential element in any kanban system looking to provide the ability to change and improve. In our project we had two colleagues from our company who we reported our progress every week and showed them the board so they could give us feedback about the work done.

4.2 Working Methodologies: Feature oriented programming (FOP)

Feature oriented programming is a programming paradigm to develop software in an incremental way. It has multiple ways to apply it to a project since it is based in three equations that allow us to see in advance which feature to choose and develop first.
Having said that, we thought that using that approach would only increase our development time since it adds an extra level of difficulty to the development cycle.

After thinking about it, we thought of using the methodogy approach of prioratizing certain features and implementing those first. But we didn't use any mathematical equation in order to do that.

What we did was, after having all the requirements completed we talked to our stakeholders and asked them which features they thought were more important and which of those would they want to get first.

Between the answers we got from our stakeholders and our own vision of how much value each feature would give to the product we created a list of priority before the development cycle had begun.

The list is the following :

1. Create groups and add/remove expenses
2. Add/Remove members from group
3. Calculate the debts of the group as well as the debts of each group member
4. Create Payments in order to pay off debts
5. Login and Create an account
6. Change user credentials
7. Upload Image for groups and users
8. Config app properties

Once we had this list, the rest of the process was easy. We simply started from top to bottom developing each feature, but there is another problem. A mobile application is divided in two environments backend (server) and frontend (client/mobile device) and that woudl make it difficult for us to develop each functionality in both environments before moving into the next one because most of the functionalities have dependencies with others.

In order to solve that problem, we separated the methodolgy in two parts, we would still use feature oriented programming but we'd first develop the functionalities of that complete list at the backend environment (because is the most time consuming) and once those were finished, we would develop the functionalities in that exactly same order on the frontend environment. All of this while keeping a clear track of the work flow in our Kanban board.

5 PLANNING

To develop this project from beginning to end, a planning has been prepared which mentions the tasks that must always be completed. These tasks have been divided to make it easier to develop. To carry out this planning, a Gantt Chart has been created where these activities are established where the deadlines of each task add up to the project deadlines.

Broadly speaking the Gantt is made of three main blocks which encompasses almost all the activities.

These blocks are:

- **Research & Planning**: Carry out a preliminary study on the project to look for other similar applications and try to figure out how were they made so we have some additional requirements.
- **Design**: Create an initial design based on the requirements obtained from our stakeholders and our team, such as MSA or scalable software.
- **Development**: Develop and deploy the project.

The Gantt chart of this section can be found at the appendix of this document.

6 DEVELOPMENT CYCLE

6.1 Requirements Gathering

Before starting the Design and Implementation stages, it is necessary to carry out a preliminary and exhaustive analysis to determine the requirements that the project must meet. With this stage we are able to reflect with clarity and precision the different characteristics of the system (requirements) classified between functional requirements (what the system must include) and non-functional requirements (system constrainst, performance requirements).

For the reasons mentioned above, we started conducting an investigation stage that would be later called **Research & Planning** where we would try to find requirements for our project in other similar applications and through meetings with our stakeholders.

We had a ten to fiveteen minute meeting with about twenty guys that we considered potential targets of our app (people between 20-35) and came up with the mentioned list of requirements that would later become the list of prioritazed features..

6.2 Application Design

Talking about the design we can distinguish three different main parts:

- Database or Model Design
- Front end Design or UI
- Backend Design or Architecture

In this section will look at the Model Design and the Backend Design and we'll leave the UI for the results section of this document.

6.2.1 Database/Model Design

Given that our current technological stack is based on reactive programming **[6]**, we had no more options that to use a non SQL **[7]** Database. The reason being that typical SQL Databases don't support non-blocking calls (which is the base of reactive programming).

After looking at multiple non SQL option like Cassandra and Couchbase we endend up deciding to work with MongoDb **[8]**.MongoDb is a non SQL **[9]** open source document based Database that has grown in popularity over the last years due to it's easy quey language, scalability and speed.
Mongo offers us these benefits over a relational database like PostgresSQL **]10]** or MySQL **[11]**.

The benefits are the following :

- Designed to be decentralized, works well in distributed systems.
- Easier to adapt to project necessities since it isn't as restrictive as a traditional SQL.
- We can change the Database Schema without having to stop the whole development (adaptability).
- Horizontal scalability, they can grow by multiplying the number of machines instead of the machine's specs.
- Queries optimized to work with great volumes of data.

MongoDB schema is organized in Collections (a list of same type documents). In the following image we can see the Collection we've created with the document schema and fields.

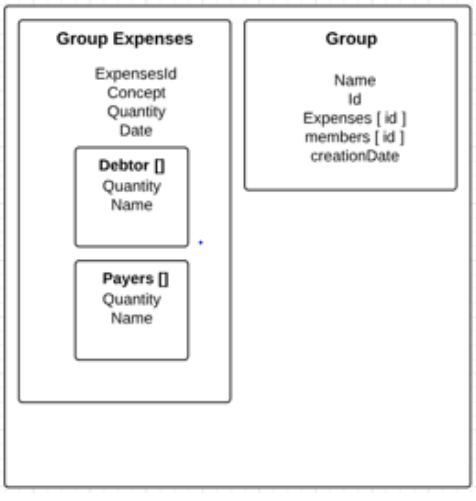


Fig 1: MongoDb schema

The result stored with this Collection schema is the following.

Fig 2: MongoDb schema visualization

As we can see, the objects are store according to the schema, but that's not something mongo does for us. We said before that mongo treats it's data as Json, we can select types for each document on the Database but it's the developer's job to ensure consistency thoughout the app since mongo won't raise exceptions if some data is not provided when we insert an object.

6.2.2 Microservice Architecture

One of the most common trends in Software Development is the use of services that provide the user what he asks for. It has come to a point that almost every major framework has adopted this aproach (Spring, Angular, Django ...).

Due to the necessity to remove the dependencies formed between these services that are generated at development and deployment **the microservice architecture (MSA)** was created.

Microservices are a new way to design and implement distributed systems where each service is completely independent from the others (normally done with Docker).

Obviously, everything comes with some benefits and drawbacks and here a fer of both.

Benefits of using microservices:

- **Scalability**: since the project is composed of small modular pieces of code that we can easily scale to multiple instances of the same service.
- **Fault tolerance**: since the project is modular, if one service becomes unavailable for some reason the rest of the system can keep working without problems.
- **Auto-Healing**: when a service becomes unavailable, the circuit breaker will redirect all incoming calls while he tries to reboot it.

Drawbacks of using microservices:

- **Complexity**: a microservices architecture forces the developers to have a deeper knowledge about deployment, testing and monitoring since there will be multiple errors as the project starts.
- **Paradigm**: People coming from a monolithic architecture will have a hard time adjusting to such a new paradigm.
- **Bug Fixing**: Finding a problem in a chain of business activities when there is a logical error can be way more complex than with a regular monolithic approach.

Now that we know about the benefits and drawbacks of the microservices architecture, we can start looking into how do they work.

Here we have the four microservices that form our architecture. Later on, we'll explore what each and every one of them exposes but for now, how does the debt service know if the user has been authenticated, or the expenses service which group to add the expense to.

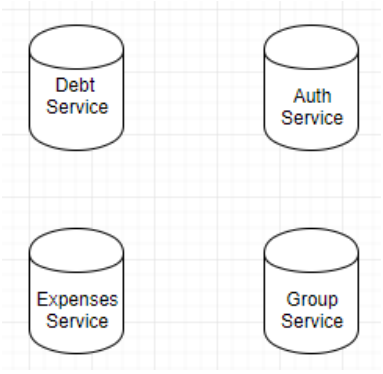


Fig 3: Microservices of WhoPays

The answer to all those questions is, via the JMS **[12]**

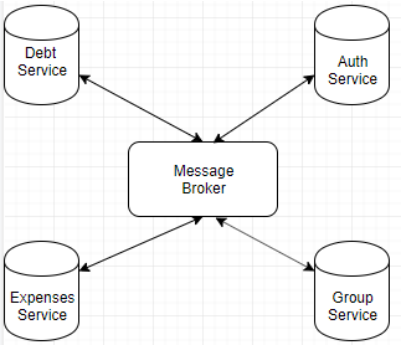


Fig 4: Microservices with JMS

The Message Broker (in our case ActiveMQ), is a implementation of the JMS that provides asynchronous communication between microservices. Message Brokers like ActiveMQ **[13]** are mostly server programs running some advanced routing algorithms.

Each microservice connects to a broker via a service called Publisher to send information and Subscriber to receive it. Messages are temporary stored in queues with a specific topic so the receiver only gets the messages from the topics which he has subscribed.

ActiveMQ allows us to create a queue architecture where we define where our services will store and retrieve data from. We can see below, the queues that our architecture has.

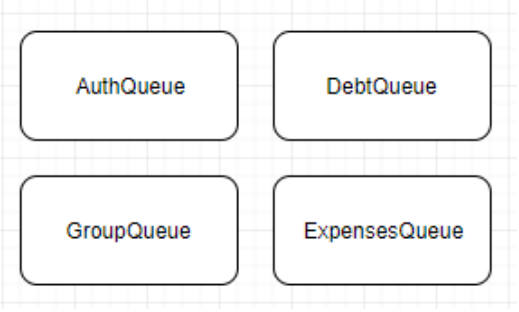


Fig 5: ActiveMQ queues

Here we have a simple example of how two microservices communicate.

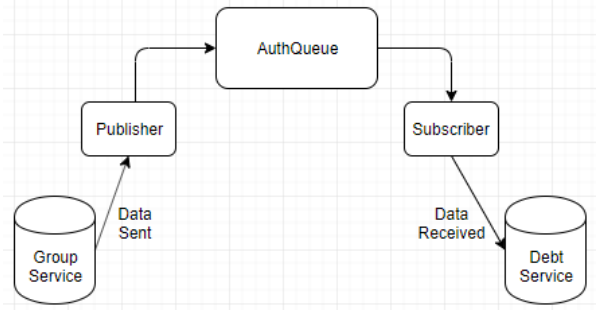


Fig 6: Communication between services

What we can see in the diagram above, is the communication that's happening when we try to create the economic balance of a specific group.

Even though we can only see here the response, previous to this diagram the Debt Service sents a message to the group service to indicate which group he needs the balance from and the Group Service returns the data according to that request as we can see.

Now that we know how a microservice architecture works and communicates between each other, lets explore what each microservice of our system has to offer.

Each service is a module that provides an API with different endpoints and methods that may have multiple puposses.

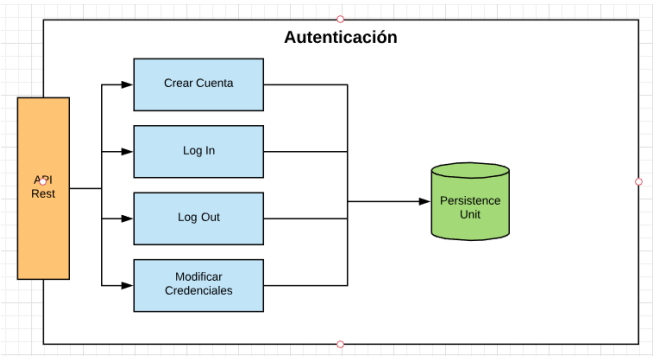


Fig 7: Auth Service

Auth Service as we can see above, provides four methods

- **Create Account**: allows the user to create an account.
- **Log In**: checks the credentials and authenticates the user.
- **Log Out**: logs the current user out of the application
- **Update Credentials**: allows the user to change his credentials (username, password).

Auth service is the one in charge of securing the access to the whole system.

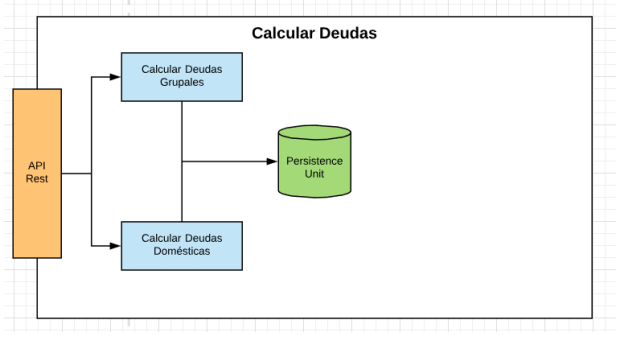


Fig 8: Debt Service

Debt Service, is the service we can see in the image. It provides two main functions but only one could be implemented.

- **Compute Group Balance**: creates the data structure of the balance of each user of the group to know who owns money.

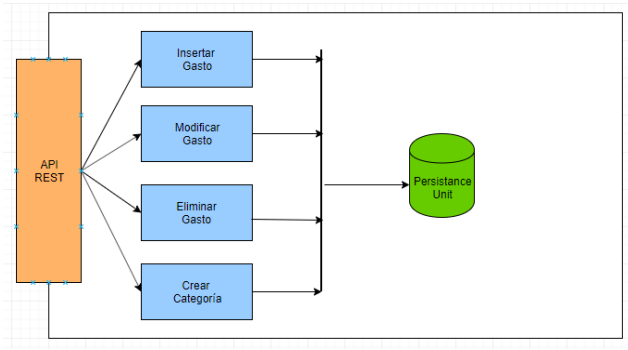


Fig 9: Expenses Service

Expenses Service is the microservice above. It is in charge of managing the expenses throughout the whole system. This service is the most complex and would be the most difficult to implement due to the complexity of adding Speech Recognition to it.

It provides the following methods:

- **Add expenses**: allows the user to add a new expense to a determined group.
- **Update expenses**: allows the user to change every aspect of the expenses in case of mistakes.
- **Remove expenses**: allows the user to remove a specific expense.

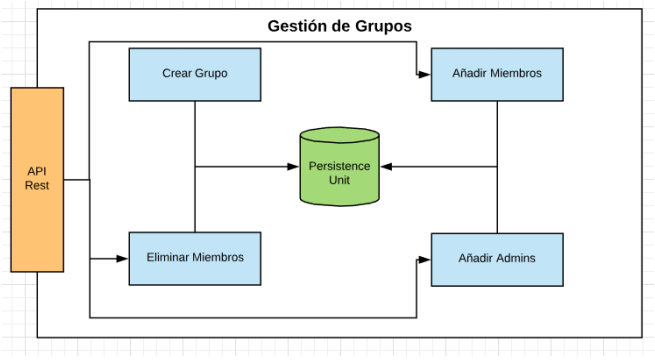


Fig 10: Group Service

The service we can see above is **Group Service**. It is in charge of managing the groups of the whole system. The service provides the following methods:

- **Create Group**: allows the user to create a new group.
- **Add Members**: allows the user to add members to a specific group.
- **Add Admins**: allows the user to make another member an admin of the group.
- **Remove Members**: allows the user to remove members from a specific group.

6.3 Implementation

Once the Design stage was finished, the Development stage started and with it the long awaited time to start implementing this architecture.

We started to implement the project from the backend environment. We needed a Persistance Unit (Database) for every service that would be eventually developed. So we started the project by creating four clusters of MongoDb from their **DbaaS [14]** (Database as a Service) at MongoDb Atlas**[15]**.

As we can see at the image above, we have here three shards of one of the four clusters that we currently have at **MongoDb Atlas**. Shards are replicas with the same dataset of the others that can be used as a backup in case of failure.

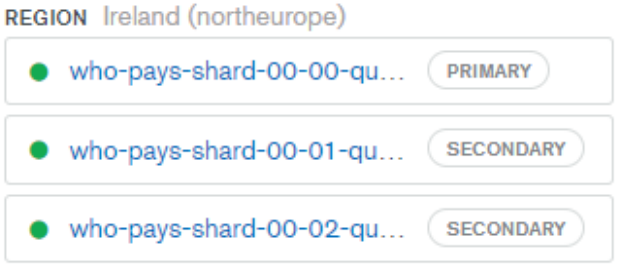


Fig 11: Mongo Db Shards

Once we had our database clusters up and running on MongoDb Atlas, we could start implementing the project on Spring Webflux.

Given the difficulty of some services and the fact that we had to implement the Ionic app, we decided to implement the platform first from a monolithic approach in order to make it easier to implement each functionality while we were migrating later.

In **Spring Webflux** is the newest version of Spring that is based on **Project Reactor**. This latest version works completely with **Reactive Programming**.

Reactive Programming is a declarative programming paradigm that has **data streams** and **propagation of change** as it's base.

In a reactive streams environment exists the following elements:

- **Publisher**: also called Observables. This objects are the ones that emit data.
- **Subscriber**: also called Observers. This objects are the ones that are notified when the data emited by the Publisher changes.
- **Subscription**: Is an event that is created by the Publisher and shared to the Subscriber.
- **Processor**: can be used between the Publisher and the Subscriber to perform maps.

Spring Webflux adds two of it's own publishers, **Mono [16]** and **Flux [17]. Mono** implements a Publisher and is used when we expect to return from 0 to 1 element.

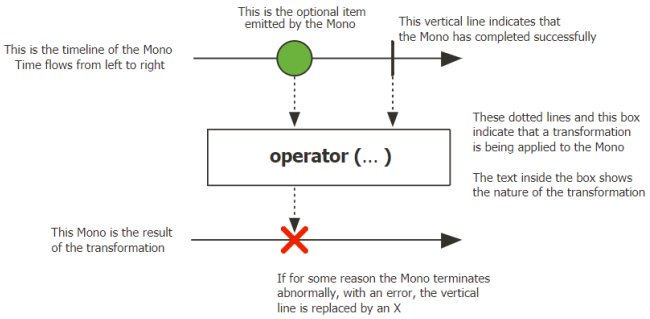


Fig 12: Mono Operator

Flux implements a Publisher aswell, but is used when we expect N elements.

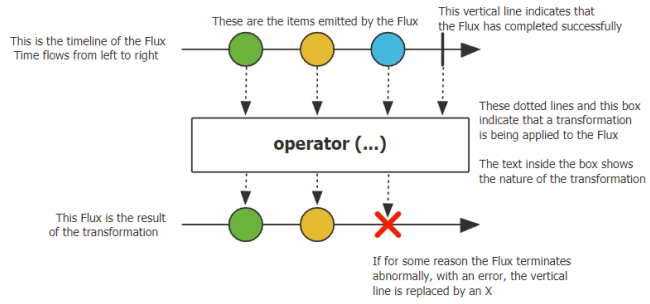


Fig 13: Flux Operator

We will now show the architecture used to build both the monolithic approach as well as every microservice.

One of the particularities of Spring Webflux that comes with being based in **Reactive Programming** is using **Non – Blocking I/O.**

Non-Blocking I/O consists of performing operations or calls without the thread having to wait for a response. The difference between a blocking or non-blocking is the following.

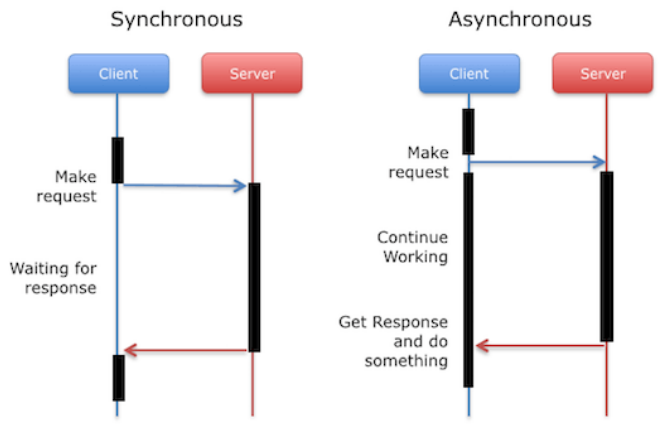


Fig 14: Asynchronous programming

We can see clearly in the image above that the client can keep doing operations instead of having to wait for a response in Synchronous (Blocking I/O).

In order to be able to use Asynchronous calls to our server, our services use Netty a Non-Blocking server instead of the standard Tomcat which is Blocking.

Spring Webflux bases his own architecture separating the project on different layers. These layers can be **Controllers**, **Services** or **Repositories** as they are the main layers but there can also be found Command or Converters.

The **Controller** layer is in charge of receiving the request and mapping it to and endPoint that our API is exposing. It binds a specific URL to an endpoint and can perform additional comprobations (access restrictions).

The **Service layer** is where the "**Business Logic**" is. By Business Logic we mean all the specific transformations or operations that are needed by the requirements. In this layer is where normally we use Converters to go from a **Data Transfer Object** (DTO) **[18]** to **a Plain Old Java Object** (POJO) **[19]** which is the one we'll be working.

The **Repository layer** is the one in charge of interacting with the Database, it's main function is to perform **CRUD** operations to the Database.

Now that we know the foundations of a Spring project architecture, let's take a look at the code of a functional reactive programming Spring project.

To do so, we'll be following the path a request to one of our services would do. To have a more visual representation of that cycle we have the following image.

The first Step of our application where the request gets is the Router. The Router is a new feature added in Spring Webflux alognside the Handler that divides the job previously did by the controller in two.

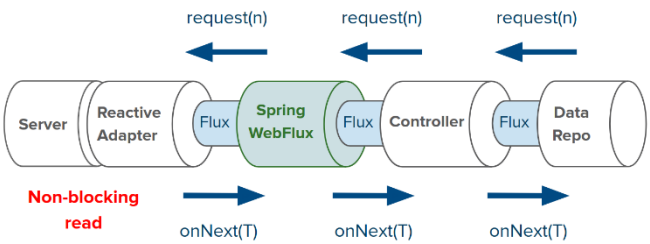


Fig 15: Request Stages

The Router will bind a URL and a protocol to an exposed endpoint and it will pass the request to the specific Handler that will do what's required.

This adds another level of abstraction where the router doesn't need to know what we have to do with a request he just redirects it to someone who knows. This is basically a Design Pattern called **Chain of Responsability [19].**

Let's continue were we left it, the request comes to the router of our service that looks like this.

```
return RouterFunctions
        .route( POST( pattern: "/user/create").and(accept(MediaType.APPLICATION_JSON)),
                userHandler::createUser)
        .andRoute(GET( pattern: "/user").and(accept(MediaType.APPLICATION_JSON)),
                userHandler::getAllUsers)
        .andRoute(GET( pattern: "/user/{userId}").and(accept(MediaType.APPLICATION_JSON)),
                userHandler::getUserById)
```

Fig 16: Router Function

If we look carefully at the image above, we can see three URLs that are being binded to three handlers. For this example we'll be following the function getAllUsers.

The Router delegates into the userHandler and redirects the request to the getAllUsers method.

```
public Mono<ServerResponse> getAllUsers(ServerRequest serverRequest) {
    return ServerResponse.ok()
            .contentType(MediaType.APPLICATION_JSON)
            .body(userService.findAll(), User.class);
}
```

Fig 17: Handler Function

We can see that the return type of this function is one of the two that we've talked before (**Mono)** but when expecting multiple User objects.

The reason behind it is that even though Mono is used for return type from 0 to 1 objects and we expect a N, we are returning a single ServerResponse that will have multiple Users as it's body. We can see that when we are creating the body of the Service we call the userService method findAll.

```
@Override
public Flux<User> findAll() { return userRepository.findAll(); }
```

Fig 18: Service Layer

Once the service is called, all that's left is to do the proper transformations before calling the **Repository layer,** but since this method is pretty simple and comes out of the box from spring there's no need to do anything else.

```
@Repository
public interface UserRepository extends ReactiveMongoRepository<User, String> {

    Mono<User> findByUsername(String username);
}
```

Fig 19: Repository Layer

This is our userRepository even if it looks like it's almost empty, there's a list of methods that come out of the box from Spring by only extending ReactiveMongoRepository.

One of those methods is findById, one of the great things that Spring does for us here is we only need to declare the method and it's attributes that if we follow the naming conventiong (findBy plus Model attribute) Spring Data will implement these methods at runtime without us having to worry about it.

Now that we've had a look at a request lifecycle and we know more about **Reactive Programming** let's see the actions that happens in every step of the process.

1. First thing, a request arrives
2. The request is routed and given to the proper handler
3. The handler identifies the parameters
4. The handler creates a pipeline in order to get the Data and returns it (**Non-Blocking)**
5. The execution environment (Spring Webflux's event loop in this case) registers a Subscriber (creates a subscription) to a Publisher (Flux in this case)
6. The Publisher (Flux) starts asking for the Users data
7. The Publisher gets the data he's asking for.

As a note, when we are working with **Flux**, we get the whole list of Users at the same time. The **Publisher** asks the Database for one User object at a time, the difference with Blocking operations is that in the meantime that we are collecting the whole list, we can keep working on other requests on the same thread.

7 TEST

Due to the lack of time and the tight schedule we have not been able to test as much as we would've wanted to. So in order to have some tests in our project we decided to focus all the test on the side of the project that we knew we could test the most with less time and resources.

Makes use of the knowledge acquired in the Software specialization we decided to use Junit to perform some Unit Test in our project. Following the pattern of examples with the code, we'll show the tests by layers.

```
@Test
public void verify_routFindAll_onCorrectRequest() {
    webTestClient.get() RequestHeadersUriSpec<capture of ?>
            .uri( s: "/categories") capture of ?
            .accept(MediaType.APPLICATION_JSON) capture of ?
            .exchange() ResponseSpec
            .expectStatus().isOk() ResponseSpec
            .expectHeader().contentType(MediaType.APPLICATION_JSON)
            .expectBodyList(Category.class) ListBodySpec<Category>
            .hasSize(3);
}
```

Fig 20: Router test

Here we can see, that we create a test request to the categories endpoint and check afterwards if the size is correct.

But, if we want to test other layers that depend on getting data from the Database we use Mockito to mock the Repository layer. Mockito **[21]** allows us to have complete control over what a repositroy function returns so we can later compare results.

```
Mockito.when(this.categoryRepository.findAll())
        .thenReturn(Flux.fromIterable(categories));
```

Fig 21: Mockito usage

As we can see, we create a dummy array of objects that will be returned when we call the method findAll from the repository layer.

```
@Test
public void verify_findAllCategories_returns3Elements_onRequest() {
    Flux<Category> categoriesFlux = this.categoryService.findAllCategories();

    StepVerifier.create(categoriesFlux) FirstStep<Category>
            .expectNext(categories.get(0)) Step<Category>
            .expectNext(categories.get(1)) Step<Category>
            .expectNext(categories.get(2)) Step<Category>
            .verifyComplete();
}
```

Fig 22: Service layer test

We can see clearly here how we call the method findAllCategories and with a StepVerifier we check every value and compare it with the dummy object we instructed the repository to return.

8 RESULTS

Now, the results obtained from the development of this project will be exposed.

Objective	Status
TFG-OBJ-01	Completed
TFG-OBJ-02	Completed
TFG-OBJ-03	Completed
TFG-OBJ-04	Completed
TFG-OBJ-5.1	Completed
TFG-OBJ-5.2	Deleted From Planning
TFG-OBJ-06	Partially Completed
TFG-OBJ-07	Not Completed

We can see here, that most of our starting objectives have been completed. The only one we could not complete were due to either lack of time (TFG-OBJ-6) or in case of TFG-OBJ-07 due to now having a possible way to deploy our project in any iOS device.

Regarding the mobile application, we've added the results in the annex section of the document.

9 CONCLUSION & FUTURE WORK

9.1 Conclusions

Since the beggining of this project, I believe that every stage of it has been quite positive. Starting from the basis that at first I was a bit lost regarding most of the technological stack that forms this project. I've learned some amazing technologies and explored part of Software Development that really intrigues and fascinates me.

I was also able to learn different Software Design Patterns that made may look to add complexity to the project at first but end up helping a lot. To be fair, I thought this project was going to be way easier than it ended up being, I underestimated the amount of time it'd take me to develop most of the project and planned according to that stimation which lead to having to replan a couple of times during the project. Due to our lack of experience in both management and development with these technologies, the development process has been delayed more than anticipated. We thought we could adapt to these new paradigms at a much higher speed that ended up happening and that's been one of our mistakes.

Regarding the mobile application, we think that there's room for improvement. Not only the current pages can be improved aesthetically but we think that adding some more pages like Expenses details for example would greatly improve the user experience.

As a final conclusion we think we tried to learn too many new technologies within the context of this project and that ended up backfiring us and slowing the whole development stage.

9.2 Future Work

Even though the college project has come to an end the system has not, due to the tight schedule only two of the four microservices have been migrated. In the upcoming weeks I plan to finish the migration of both the **Group Service** and the **Expenses Service** from the monolithic approach to a full microservices architecture.

Also due to a very tight schedule, we were only able to create some test cases for unit test and not integration tests. That is also something that we are looking forward to implement in this upcoming month.

Once the original vision of the project has been fullfilled our next step will be to try to integrate the project with other apps/services to make our user's experience even better. Some current ideas include get the expenses directly from the bank account, be able to pay your debt from the app itself or even implementing a group chat.

ACKNOWLEDGEMENTS

First of all I'd like to thank my tutor Lluís Gesa for the time dedicated to guide me through this project and for all the support. I would also like to thank my colleague from my previous company Gustavo Acuña for helping me designing the microservices architectures and pointing to me some advices about best practices.

Finally and most important, I'd like to thank Pau Gallardo for all the help and guidance he's given me through the whole process of this project. From best practices in both Backend and Frontend, designing the MongoDb schema, usage of Docker and specially Design and Implementation of the Ionic App. I dont know if I would've made this project without all that help.

References

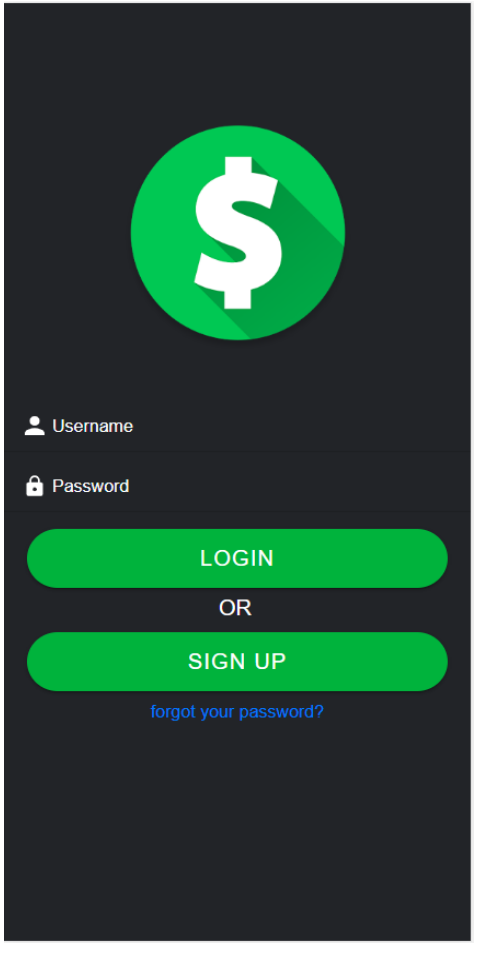
The images regarding figures 12,13,14 and 15 have been extracted from https://projectreactor.io/

[1] Ionic 4, link: https://ionicframework.com/docs, April 2019
[2] Tricount, link: https://www.tricount.com/es/hacer-cuentas-entre-amigos, January 2019
[3] Ionic Native, link: https://ionicframework.com/docs/native/overview, April 2019
[4] Speech Recognition, link: https://ionicframework.com/docs/native/speech-recognition, April 2019
[5] Docker, link: https://www.docker.com/get-started , February 2019
[6] Reactive Programming, link: https://gist.github.com/staltz/868e7e9bc2a7b8c1f754, January 2019
[7] SQL, link: https://devcode.la/blog/que-es-sql/, December 2019
[8] MongoDb, link: https://www.mongodb.com/what-is-mongodb, February 2019
[9] Non SQL, link: https://es.wikipedia.org/wiki/NoSQL, May 2018
[10] PostgresSQL, link: https://es.wikipedia.org/wiki/PostgreSQL, February 2019
[11] MySQL, link: https://es.wikipedia.org/wiki/MySQL, February 2019
[12] JMS, link : https://www.javatpoint.com/jms-tutorial, May 2019
[13] Active MQ, link: https://www.adictosaltrabajo.com/2012/07/25/active-mq/, April 2019
[14] DbaaS, link: https://www.techopedia.com/definition/29431/database-as-a-service-dbaas, January 2019
[15] MongoDb Atlas, link: https://www.docker.com/get-started, May 2019
[16] Mono, link: https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html, May 2019
[17] Flux, link: https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html, May 2019
[18] DTO, link: https://www.oscarblancarteblog.com/2018/11/30/data-transfer-object-dto-patron-diseno/, March 2019
[19] POJO, link: https://es.wikipedia.org/wiki/Plain_Old_Java_Object, March 2019
[20] Chain of Responsability, link: https://refactoring.guru/design-patterns/chain-of-responsibility, April 2019
[21] Mockito, link: https://site.mockito.org/, June 2019

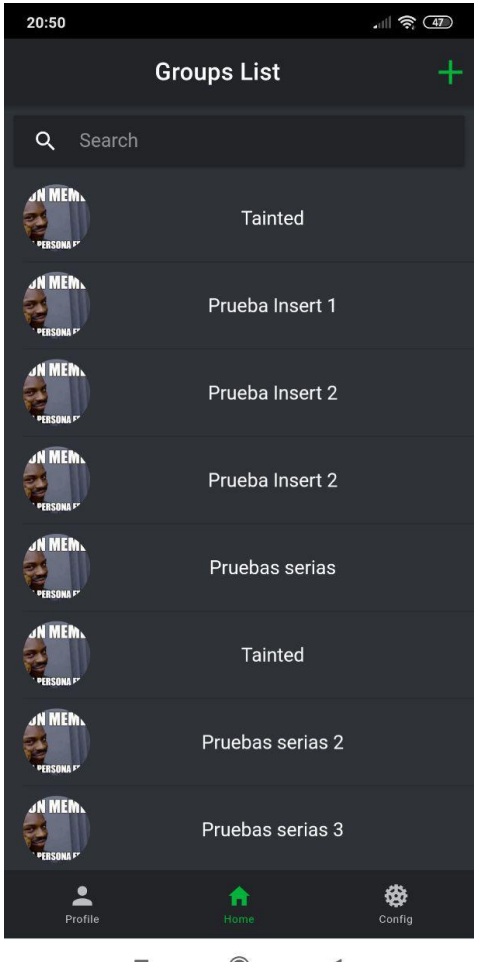
E-mail de contacto: rafa.gomezper@gmail.com
Mención realizada: Ingeniería del Software.
Trabajo tutorizado por: Lluis Gesa Boté (DCC)
Curs 2018/19

9 APPENDIX

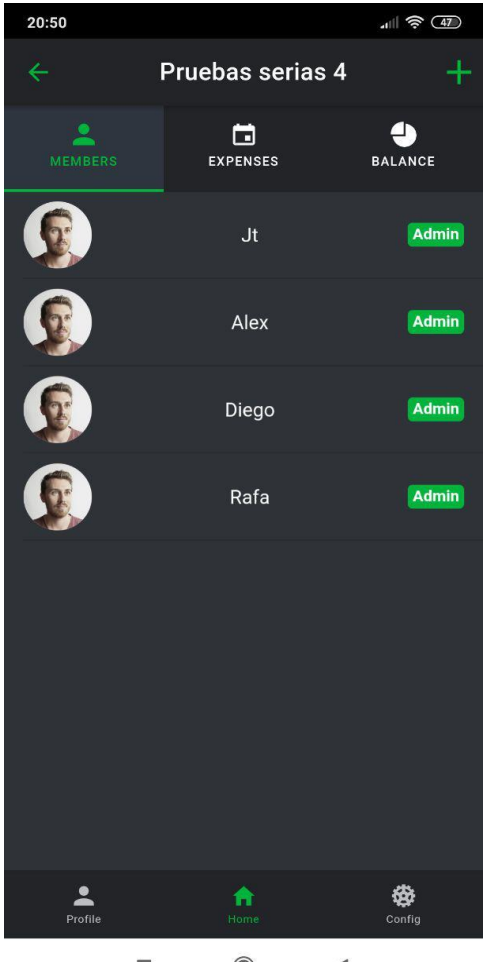
9.1. APP LOGIN



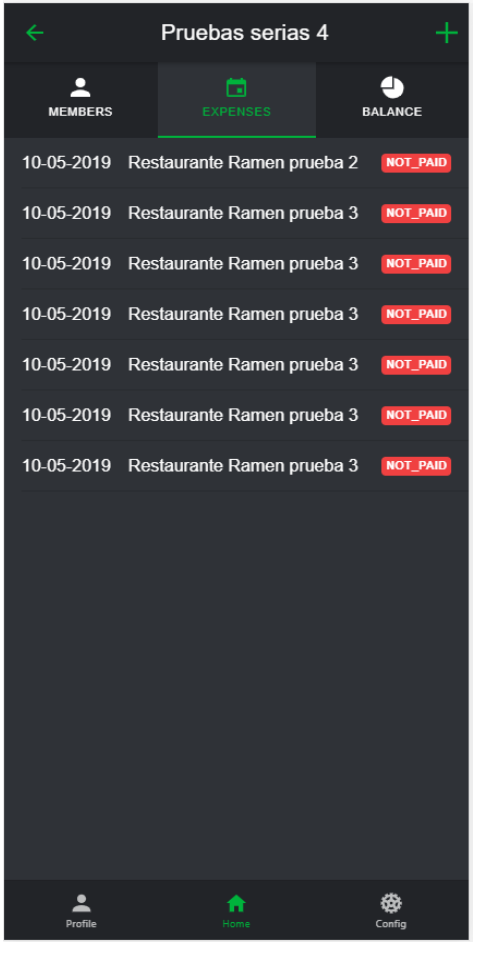
9.2. APP HOME PAGE



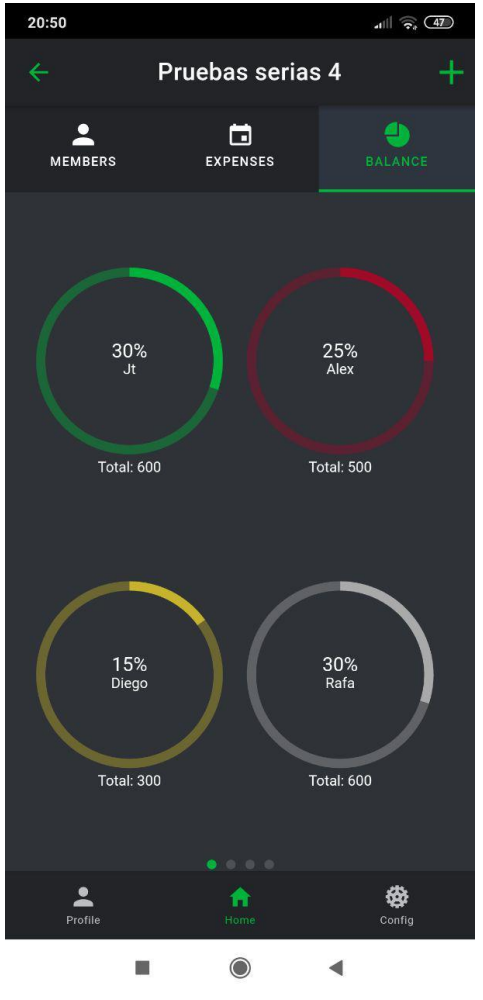
9.3 GROUP MEMBERS PAGE



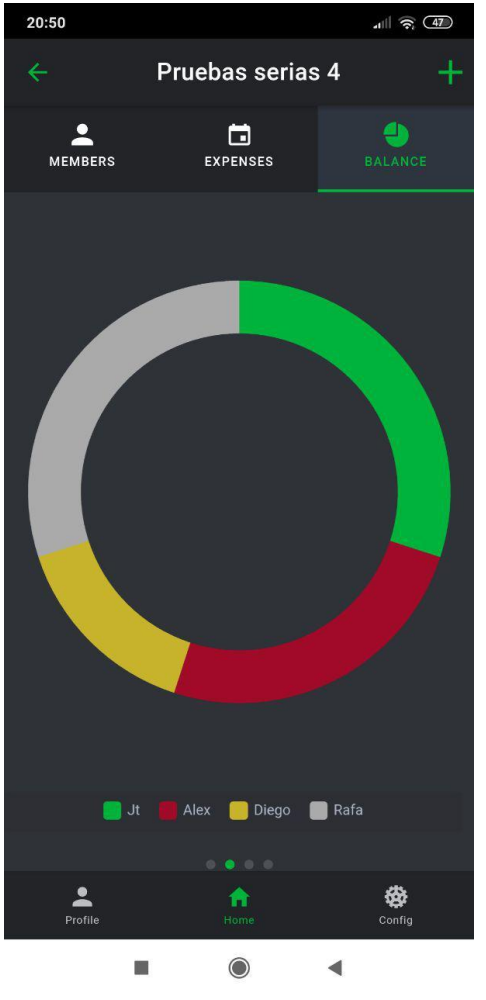
9.4 GROUP EXPENSES PAGE



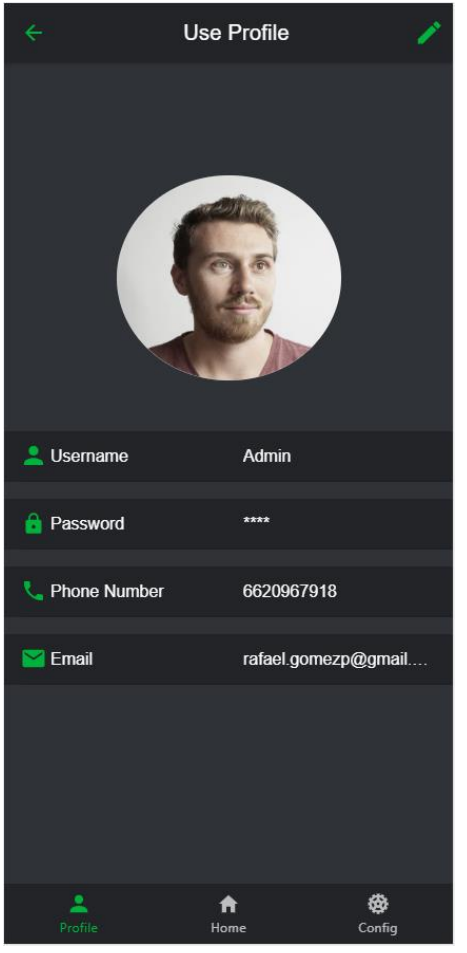
9.5 GROUP BALANCE GRIED PIE



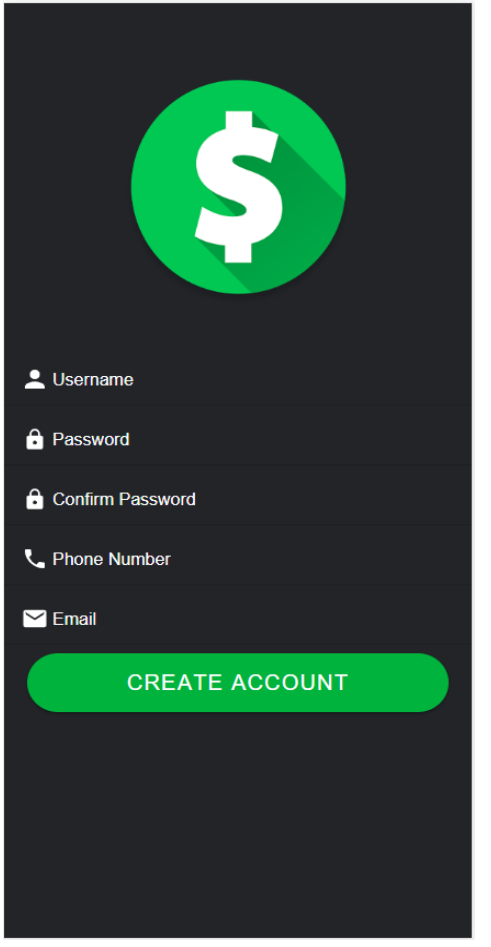
9.6 GROUP BALANCE PIE CHART



9.7 APP USER PROFILE



9.8. APP USER REGISTER



9.9 GANTT DIAGRAM