الجمهورية الجزَائِرية الديمقرَاطية الشعبية
DEMOCRATIC AND POPULAR REPUBLIC OF ALGERIA
وزَارة التعليم العَالي و البحث العلمي
MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH
جَامعة عمَار ثليجي بِالأَغوَاط
UNIVERSITY AMAR TELIDJI LAGHOUAT

FACULTY OF SCIENCE
COMPUTER SCIENCE DEPARTMENT

Master's thesis

Field: Mathematics and Computer Science

Faculty: Computer Science

Option: Computer Systems and Networks

BY:

Habib Salah Mounir **Goudjal Youcef**

Title:

On Demand Services App Following Clean Architecture .

Supervised by: Lahcen Mohamed Bensaad

President: Taher Allaoui

Examinator: Mohamed Lamine Ameur

Supervisor: Mohamed Lahcen Bensaad

Academic year 2021/2022

ملخص

يشهد قطاع التجارة الإلكترونية انتعاشا بفضل كثرة الطلب ووفرة الوسائل وكلنا نعلم مدى أهمية هذا القطاع ولاسيما في حفاظ على تدفق السلع والأموال وكذا الدور الذي يلعبه هذا القطاع. لكن بالرغم من توفر الوسائل الا انه يصعب على عامة الناس إيجاد الوسيلة دون اللجوء إلى الوسائل التقليدية ; وعليه هدفنا من هذا العمل انجاز منصة إلكترونية تسمح بتنظيم هذا القطاع والتواصل بين التجار والزبائن وكل هذا من أجل تسهيل عملية البيع والشراء بينهم وعمل بصفة أسرع من الوسائل التقليدية. وفي الاخير توصلنا لإنجاز منصة تسمح للزبائن والتجار بإرسال طلباتهم عبر الإنترنت واختيار العرض المناسب لهم وقمنا بهذا الانجاز بالاستعانة ببعض التكنولوجيات الحديثة مثل الهندسة النظيفة من أجل مساعدتنا في هذا العمل وحل المشكلة المذكورة أعلاه

Abstract

The demand for goods and food is high at all times and continues Over time, we all know the importance of products Maintaining the flow of goods and cash and its role in the economy The problem is that despite the availability of goods and food, people do not have it easy access and often have to resort to traditional means to find a suitable one. The goal of this work is to create Plat models that organize the buying and selling process and help both the seller and Potential customers to interact better and find work (for merchants) and business means (customers) easily, and the result is a platform that allows customers to post their orders Merchants bid and customers can then choose the best deals that suit them , thus solving the above problem

DIDICATES

I offer my greetings to my dear parents, who are the cause of my existence in this life for their support, their patience and their love that gave me the strength to continue my studies. Also To my brothers and sisters to whom I wish success. To all my best friends whose list is long. To all my great family. To all my friends and to all my teachers and specialy I give my regards to my partner Habib Salah Mounir Thanks for everything .

Goudjal Mohamed Youcef

I offer my greetings to my dear parents, who are the cause of my existence in this life for their support, their patience and their love that gave me the strength to continue my studies. To my brothers and sisters to whom I wish success. To all my best friends whose list is long. To all my great family. To all my friends and to all my teachers and specialy I give my regards to my partner Goudjal Thanks for everything .

Habib Salah Mounir

ACKNOWLEDGEMENT

First of all, All praise and gratitude is due to allah, for giving us the will and strength to complete this work. We would like to express our sincere gratitude to our supervisor Mr.Bensaad Lahcen, for his support and the amount of time and effort he put into guiding us through the duration of this project. We would also like to thank our teachers that have given us the knowledge required to complete this work. A special thanks to our families: both our amazing fathers and mothers, our sisters and brothers who have supported us through thick and thin, and have been an inspiration to us through our life.We could never have asked for better families. This work is dedicated to all of you .

CONTENTS

LIST OF FIGURES

GENERAL INTRODUCTION

Introduction

The world has experienced unprecedented economic development due to the rapid evolution of technology. It has made many complex and time-consuming tasks easier to perform, and has made it possible to earn more money in less time and with a minimum of effort. The field of commerce has seen great advances over time. It has since evolved into intercontinental trade, which involves massive chains of production, delivery and large-scale transactions.

Trade is what drives the global economy, as people have always done. They are attracted to places where business is good, looking for opportunities. The goods involved in trade were limited by distance and volume The type of goods traded and the lack of appropriate mechanisms for this overcome these limitations. In the past, people transported everything in horse-drawn carts and it took several weeks to deliver the goods to their destination, but this is no longer the case, as things continue to evolve. The evolution of commerce to what we know today is due to the revolutions systems that have been made in the field of electronic commerce in its various areas, and this can be clearly seen with cell phones that serve the needs of people without moving Used for multiple purposes, they save a lot of time and money in the process.

problem

With a new innovation come new challenges that did not exist before. As the demand for e-commerce is high at all times and continues to grow, the number of websites and applications is growing rapidly as well. The problem remains the inability to find work for merchants, store owners, and transportation to customers at the local and national levels.

Like supply and demand are equally present and identical but the connection between them is the most difficult part. For example, a customer may have an order for example pizza or candy, but neither knows the other, so neither benefits. Then the customer has to use traditional methods to find out how to get their order (phone number or inquire)

Solution

Our Solution is to design a platform where merchants list their products and foods, Customers can browse them and choose what they like. They can also post job offers, then customers can choose the offer they make Both parties can then contact each other via the platform to negotiate. Prices and other related items with clean design and clean architecture.

Goal:

the aim of this project is not to just develop an app but more interestingly is learn how to implement fast and secure communication between backend and frontend , and Olso learn learn how implement a system following the principle of clean architecture and modern design patterns

CHAPTER 1

ECOMMERCE GENERAL VIEW

1.1 Online Commerce (E-commerce):

Definition :

Online commerce or e-commerce is the monetary exchange of goods, services or information via computer networks, particularly the Internet. In the context of business-to-business trade, merchants have been using electronic data interchange (EDI) networks for many years. Electronic transactions are also conducted over cell phone networks. This mobile commerce is called mobile commerce. In a context of strong environmental constraints, the development of distance selling tends to transform the logistical problems linked to the world of commerce. The term online commerce also includes the global circulation of data

1.2 History:

he emergence of online commerce is directly linked to the appearance of the Web in the early 1990s. On August 11, 1994, Phil Brandenberger, a Philadelphia resident, placed the first online order using a secure credit card payment system. The New York Times covers the event and points out that behind a small click for an individual is a big step for the economy. This first purchase of 12.48 for a Sting album represents the first stone of an edifice that has been growing ever since. In France, online commerce was first developed via the "Minitel" on which were present big names in mail order, such as La Redoute or Les 3 Suisses6,7. The arrival of the Internet quickly led to the development of a different business model8. In the late 1990s, this business model was made famous by Amazon, EBay and AOL, companies that took advantage of a market

capitalization bubble of young companies without precedent in history, which ended in a crash, a phenomenon that also affected many small biotech companies . On June 8, 2000, the European Parliament and the Council adopted a European directive on e-commerce, named Directive 2000/31/EC of the European Parliament and of the Council on certain legal aspects of information society services, in particular e-commerce, in the internal market. This directive was transposed in France by the 2004 law on confidence in the digital economy. This French legal framework concerning the collection of nominative information intended for commercial prospecting is heavy [According to whom?]. Indeed, the exemptions to the principle of prior consent are extremely restrictive for prospecting individuals and professionals by email . To comply with this law, file sellers often use service providers located outside the national territory since French law does not apply there . The arrival of mobile telephony has introduced a new rupture, with a quantitative pricing per data, we also speak of mobile commerce.

1.3 Related work:

In this section, we will take a look at some related works, and at the end We will make a comparison between them.

Ouedkniss : Seriously, I think we don't even need to explain what Ouedkniss is! Ouedkniss.com allows you to sell and buy in your city, and everywhere in Algeria. Post your classified ad for free and arrange to have it delivered directly to the buyer. Free classified ads of individuals and professionals who offers online ads to buy or sell a house, rent an apartment, but also used cars, utilities or scooters. You will also find job offers or services, pets or computer equipment, multimedia products, fashion and beauty. Place a free classified ad. [20]

Figure 1.1: Ouedkniss website mobile view

Uber Eats: Uber Eats is part of Uber's suite of apps and specializes in meal delivery. Uber, it has separate apps for drivers and customers. [21] It works as follows:

- After logging in, customers are greeted with a pre-selected list of restaurants.
- Customers can choose a restaurant from this list or search for the restaurant of their

choice.

- or search for the restaurant of their choice. After clicking on the restaurant, customers are invited to choose from the restaurant menu.
- They then place their order
- The driver closest to the passenger then receives notification of the order and can choose to take it or not.
- order and can choose to take it or not.

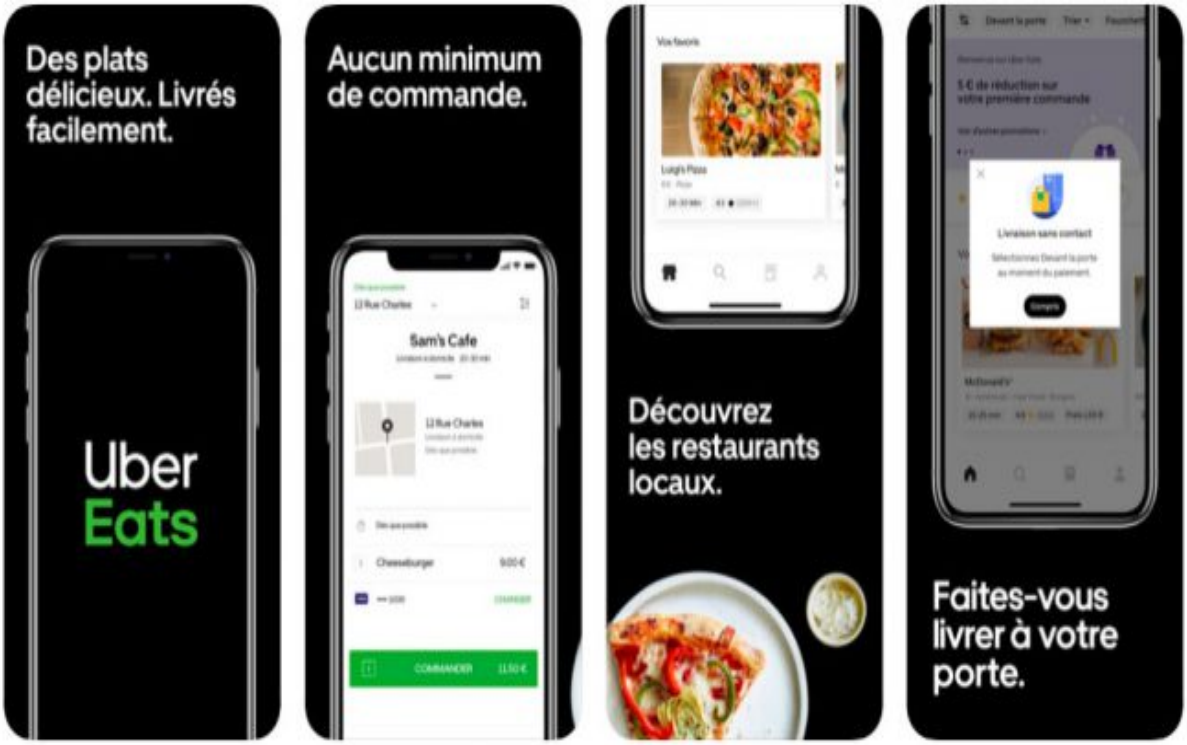


Figure 1.2: Uber Eats android app

Jumia Food: Jumia is an online retail company with a presence in the African market. Founded in 2012 by two Frenchmen, Jumia sells electronics, hygiene, food and services.2 Jumia's platform is an online marketplace that connects sellers and buyers, providing them with a logistics service, enabling shipping and delivery of packages in addition to a payment service.1 In 2019, more than 80,000 sellers offer a wide range of on-demand products and services3 : household appliances and electronics, fashion, children's toys, but also services such as hotel or airplane reservations, and meal delivery. Jumia has been called the African Alibaba or the African Amazon. [22]

Figure 1.3: Jumia Food android app

1.4 Conclusion:

In this chapter, we have discussed e-commerce and its importance. Focusing on the food aspect We have also seen statistics that show us the evolution of this field. Finally, we have seen some works related to our project. in the next chapter, we will define and Analyze the functionality of our platform

CHAPTER 2

UML DESIGN AND CLEAN ARCHITECTURE

In this Chapter, we examine the design of our work in the form of conceptual diagrams. We used UML for our project, because it provides a set of diagrams that are the easiest to implement and translate into actual code in our case and Clean Architecture :

2.1 UML:

UML or Unified Modeling Language is a standard language for specifying, visualizing, building and documenting software system artifacts. It was created by the Object Management Group (OMG) in 1997 and is now part of its standards. UML is used to model the structure as well as the behavior of systems, software or not. It is used to represent various aspects of the constitution and behavior of a system:[19]

2.1.1 Structural diagrams:

As their name suggests, they describe the structure of a system, there are 7 structural diagrams in total. in total :

- Class diagram : Commonly used in OOP (Object Oriented Programming), a class is composed of a name, attributes that describe its structure and operations that give a general idea of its behavior. The relationships between classes are represented by the connecting lines between them. The combination of classes and their relationships forms the class diagram.
- Object diagram : An object diagram is essentially an instance of a class diagram. It is used to remove the abstraction of the class diagram and test its structure in practice. in practice.

- Component Diagram : The component diagram is used in the documentation of complex systems. The component diagram is a way of breaking down smaller components, which provides a high-level view of the system. It also helps to isolate the system's functionality for better understanding.

- Composite structure diagram: It represents the internal structure of a class and the interactions between the different components of the class.

- Deployment Diagram: This diagram is used to visualize the deployment of software components on the hardware.

- Package diagram: The package diagram is used to represent the relationship between the different macro components of the system. macro-components of the system. It essentially does the same thing as the composite structure diagram, but on a larger and opposite scale. composite structure diagram, but on a larger and opposite scale.

- Profile diagram: This type of diagram is used as an extension of the UML itself, so you can customize it and add new concepts.

2.1.2 Behavioral diagrams:

These diagrams describe the behavior of the system and, like the structural diagrams, there are 7 in total. There are 7 in total:

- Use case diagram : This diagram is used to represent the general operations and required functionality of a system. A use case diagram has three main components: - Actors: represent the parties involved in the interaction with the system, it can be a human, a group or an organization. - Use cases: circled verbs that represent the functional requirements of the system. - Relationships: between use cases, such as extension or inclusion.

- Activity diagram: This diagram is used to describe the flow of activities and operations of the system in the context of the objects involved.

- Interaction Summary Diagram: This is like a specialized activity diagram that is an activity diagram of other interaction diagrams.

- Synchronization diagram: This diagram is used to represent the interaction between different objects over time. objects through time.

- State Machine Diagram: Used to represent the states of an object and how it is affected by internal or external factors (events). Internal or external factors (events).
- Communication diagram: It focuses on the communication between different objects in the system.
- Sequence diagram: This diagram describes the behavior of the system in detail as well as the state of the objects and actors involved. Use cases in motion [8]

We have used 3 diagrams in our thesis, one structural which is the class diagram and 2 behavioral ones, which are the use case and sequence diagrams

2.2 Use Case diagram's

In this diagram, we have the main use cases(The actions he can do in the app) of our project. for example :

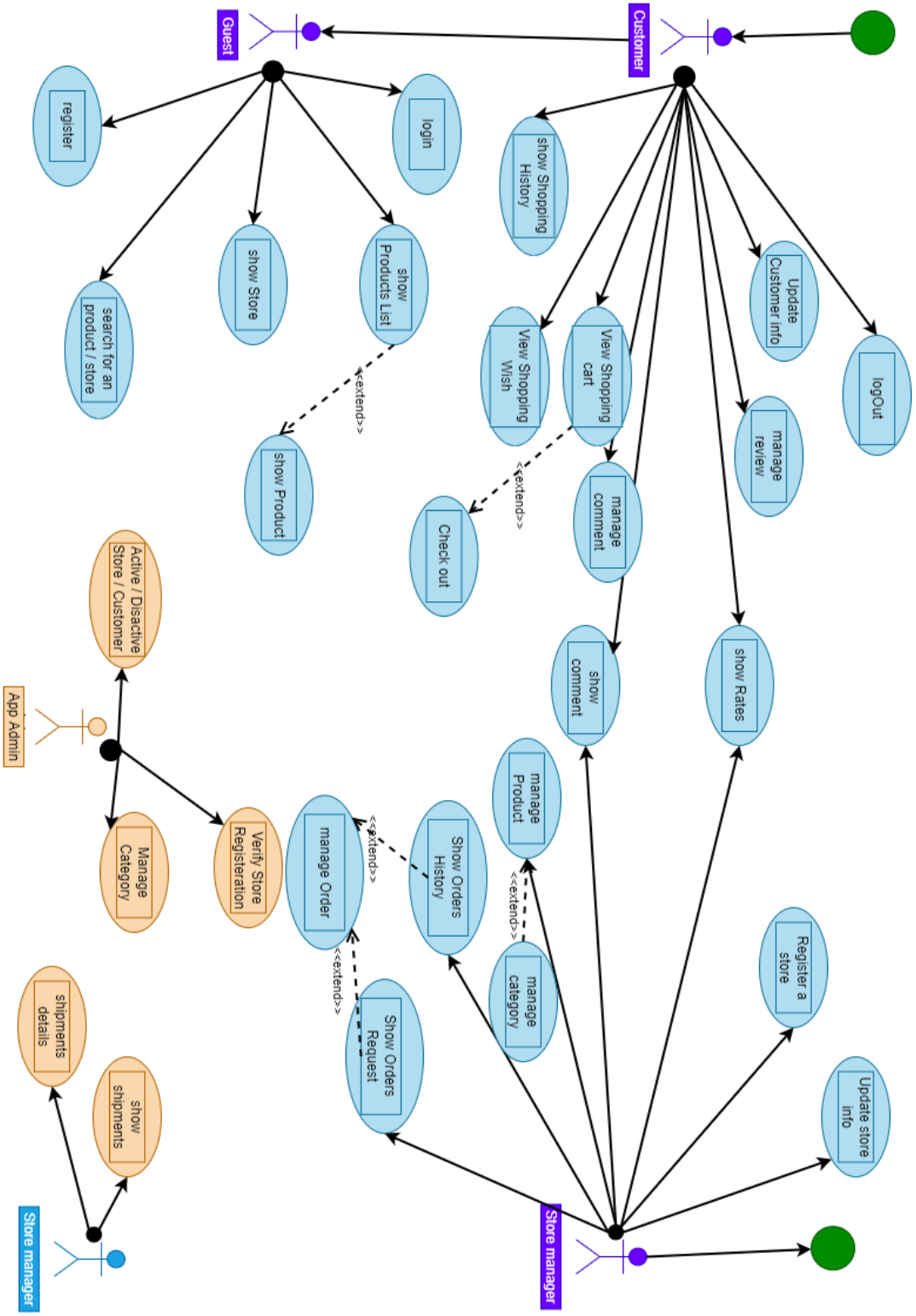


Figure 2.1: Use Case Diagram

2.3 Class diagram

This diagram will show the main classes of our project and the important fields and methods. we have :

- **Customer** : This class is used to represent a customer. It has a name, a phone number, a mail address and a password. It has a method to add a new order to the database. It has a method to get the orders of the customer.

- **Order** : This class is used to represent an order. It has a number of items, a total price and a date. It has a method to add a new item to the order. The order status is list ¡Pending, In Progress, Completed, Cancelled¿.
- **Product** : This class is used to represent an product. It has a name, a price and a description. methods are restock and update the product.
- **Store** : This class is used to represent a store. the store can add new category list orders ....

Here is our class diagram :

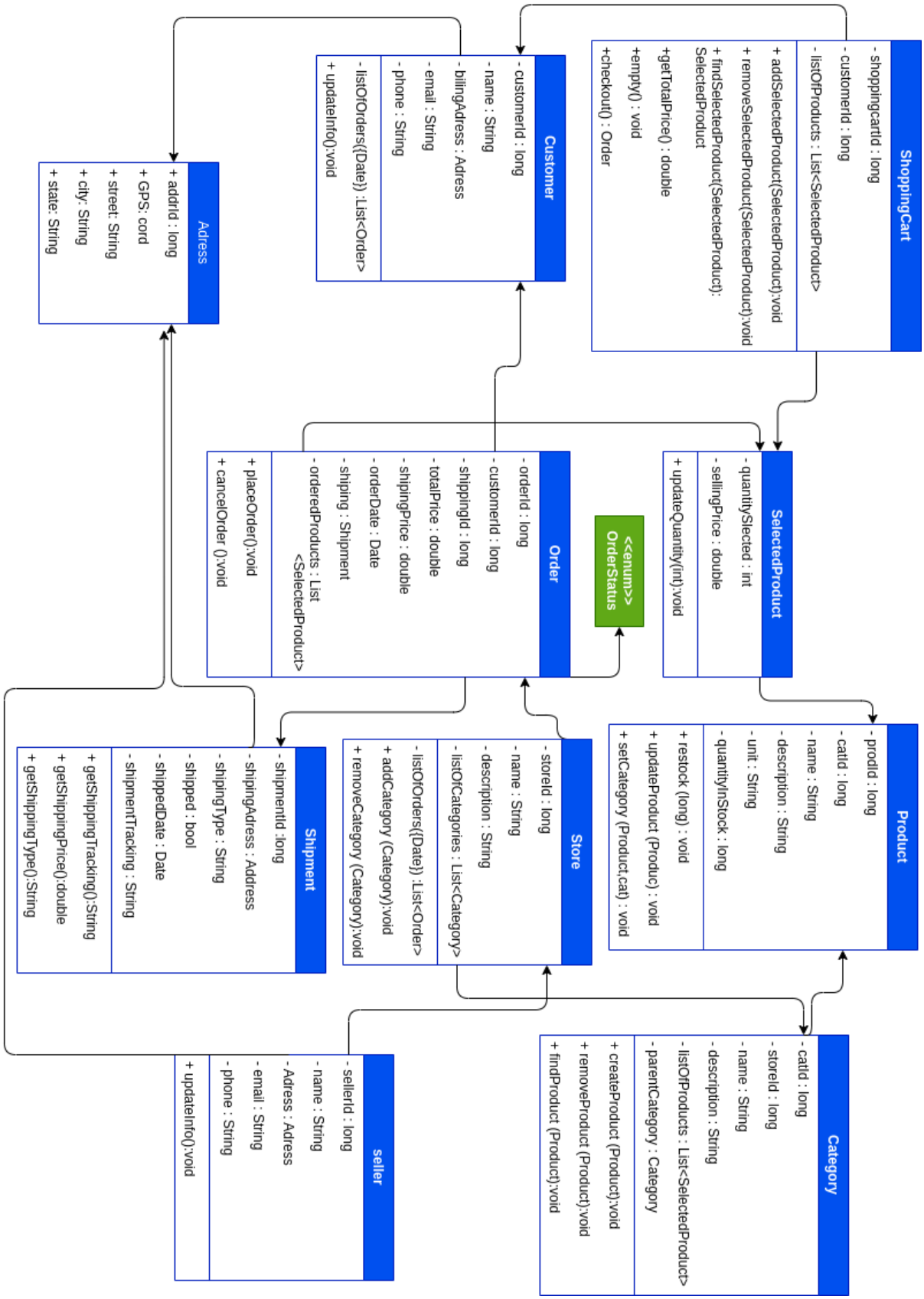


Figure 2.2: Diagram de class

2.4 Sequence Diagrams

In this section, we will see the most important sequence diagrams of our project.

- **Login** : The guest can login to the app.
- **Register** : The guest can register to the app.
- **Update Customer info** : The customer can update his info.
- **manage comment** : The customer can manage his comments. create one or delete one.
- **manage order** : The store manager can show his orders and update the status of the order.
- **manage product** : The store manager can show his products and update the status of the product.

The store manager can do all what the customer can do. And the customer can do all what the guest can do.

And in the other side, there is the app admin who can accept the requests to activate the store or to delete the store.

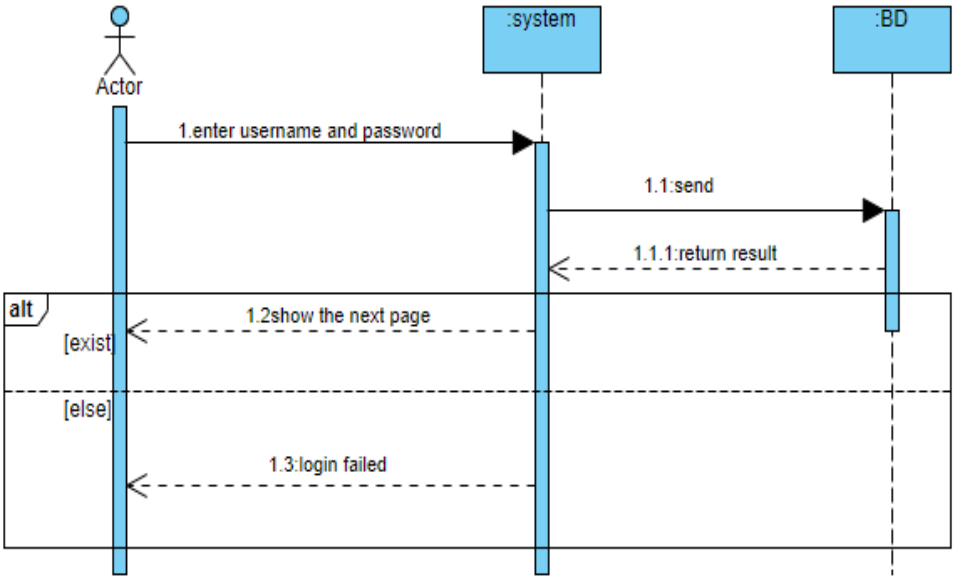


Figure 2.3: User Login Sequence Diagram

This diagrams shows the sequence of the login process. firstly the user enter his username and password and then he will be redirected to the home page or if the login failed will shows error message.

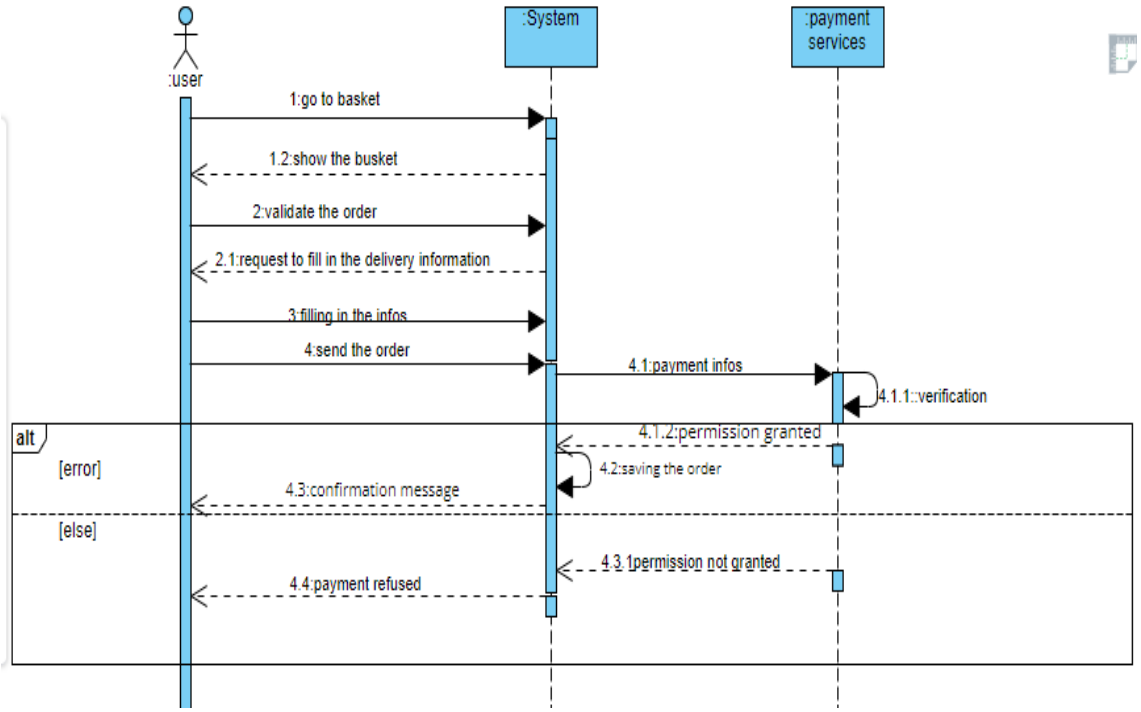


Figure 2.4: Payment Sequence Diagrams

This sequence diagram shows the sequence of the payment process. starting with the client picks items or services from the app then he validate his basket and then he will be redirected to the payment page. he has to pick one option of payments (cash, Dahabia, etc..). Now he send the order, The system will check the payment and then the system will send the order to the store.If the payment is refused the system will show error message.

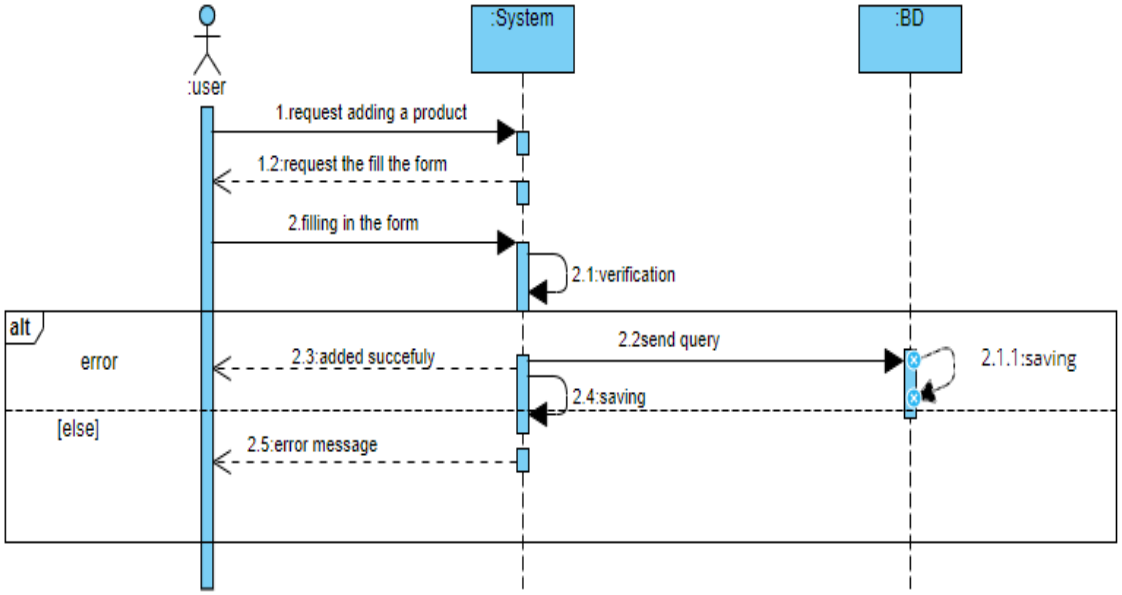


Figure 2.5: Add Product Sequence Diagrams

This sequence diagram for store manager, shows the sequence of the adding product process. starting with the store manager adds a new product with details like (name, description,price, etc) to the app. this data will be stored in the database.

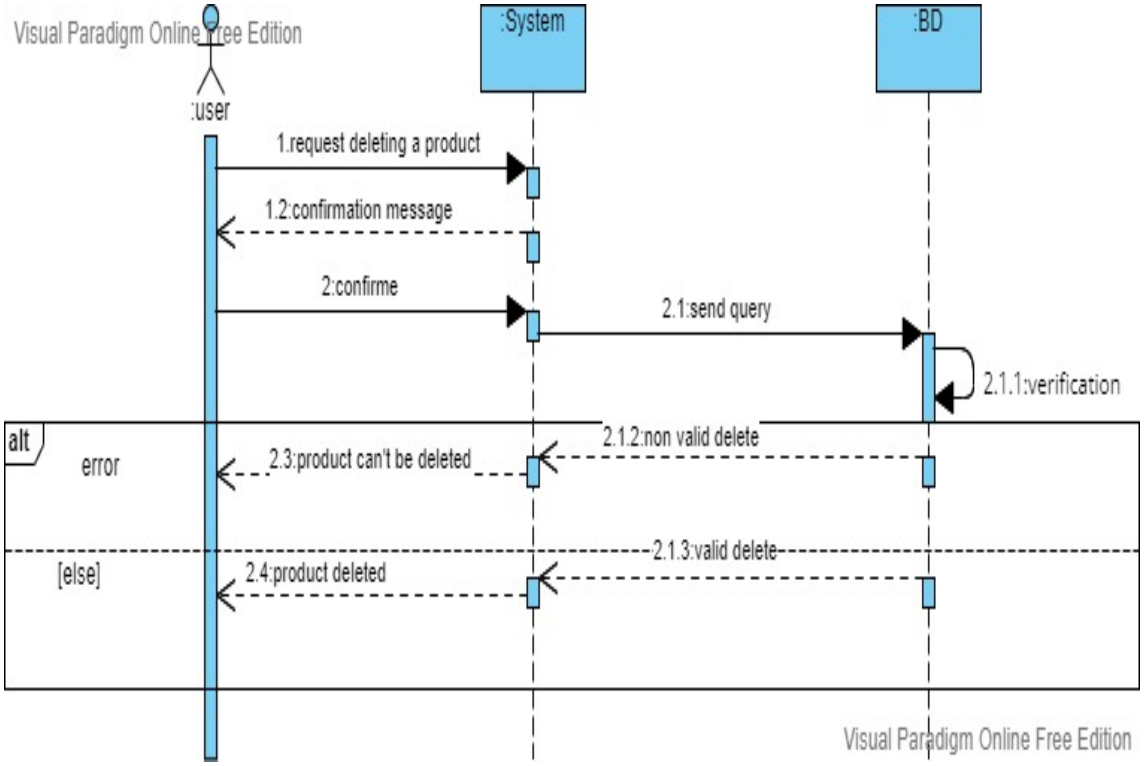


Figure 2.6: delete Product Sequence Diagrams

This diagrams shows how the store manager can delete a product from the app. By requesting to delete the system respose with confirmation requests if the user confirm the system will delete the product otherwise snack error will be shown.

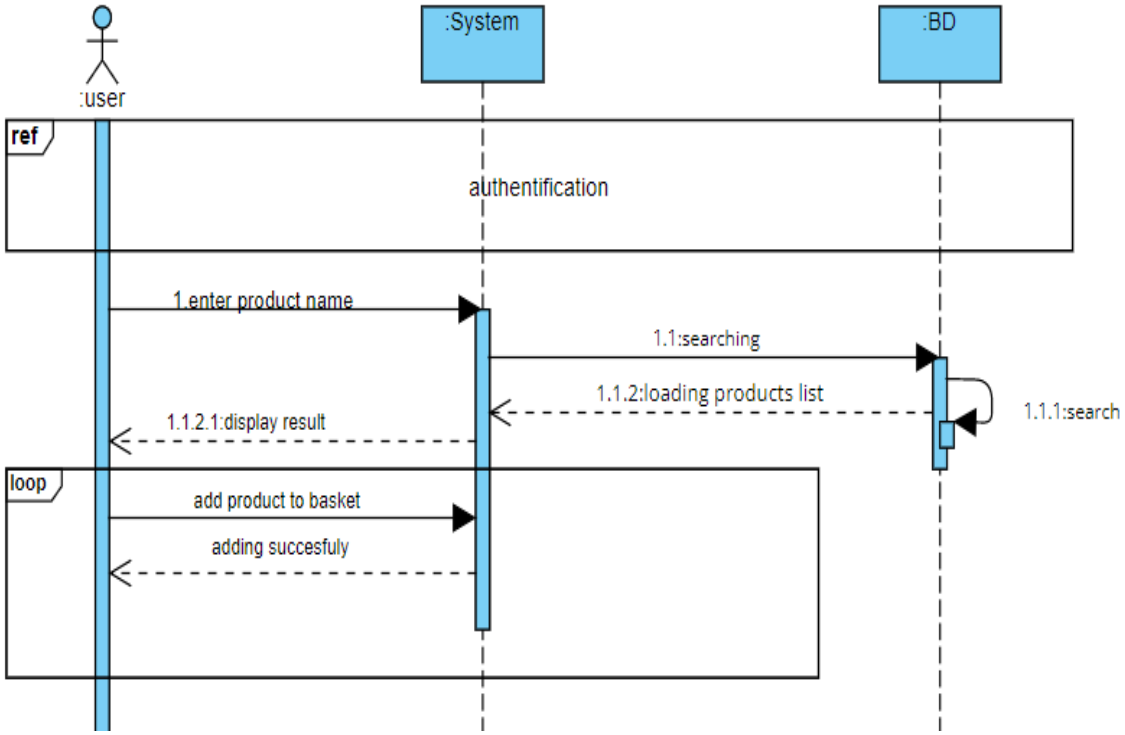


Figure 2.7: search for Product Sequence Diagrams

After the client is authenticated he can search for products in the app. searching process looks like this :
enter the product name in the search bar and the system will show the product that matches the name. main time will show the client progress loading indicator. When the system fetch the products from the database we display the result. optionally the client can add the product to his basket.

2.5 Clean Architecture:

Clean architecture is the latest book by Uncle Bob [11]. It defines architectural patterns to make software easy to change. In this section, we will go through the book main concepts and the patterns used to build the software.

2.5.1 Introduction:

What is architecture? Architecture is the process of defining the structure of a software system. Architecture supports the behavior of the system, while keeping its flexibility to change.

What is Clean Architecture? Clean Architecture is an architectural style that follow Uncle Bob's principles. The key idea is to use the dependency inversion principle to separate the software into modules. This create a "plug-in" architecture thats keeps the system flexible and maintainable.

2.5.2 Code design principles (SOLID)

A clean architecture starts from [11, clean code].

Clean code follows SOLID principles. These are guidelines to write focused, extensible and maintainable methods and classes. They are important to architecture because they help to define the structure of the software.

Here are the SOLID principles:

- **Single responsibility principle**: a class should have one, and only one, reason to change. Or the new version: a module should be responsible to one, and only one, actor.
- **Open-closed principle**: a class should be open for extension, but closed for modification.
- **Liskov substitution principle**: a class should be substitutable for its subclasses. means objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
- **Interface segregation principle**: a class should not have too many methods.
- **Dependency inversion principle**: one should depend upon abstractions, not concretions.

2.5.3 Architecture principles

Architecture is the shape of the system. It includes the system's division into modules, the modules' division into classes, and the classes' division into methods. And the ways those components interact and communicate with each other. The purpose of the architecture is to facilitate the development, deployment, and maintenance of the system.

Therefore, architecture should not only support behavior but also provide flexibility in the life cycle of the system.

Setting boundaries

Boundaries are the ways the system is divided into modules. They separate things that matter from things that don't.i.e. high-level components from low-level components. If a high-level component depends on a low-level component at the source level, changes in the low-level components will spread to the high-level component. Therefore, we place a boundary between the two, using polymorphism to invert the logic flow. This is the **dependency inversion principle** in the SOLID principles.

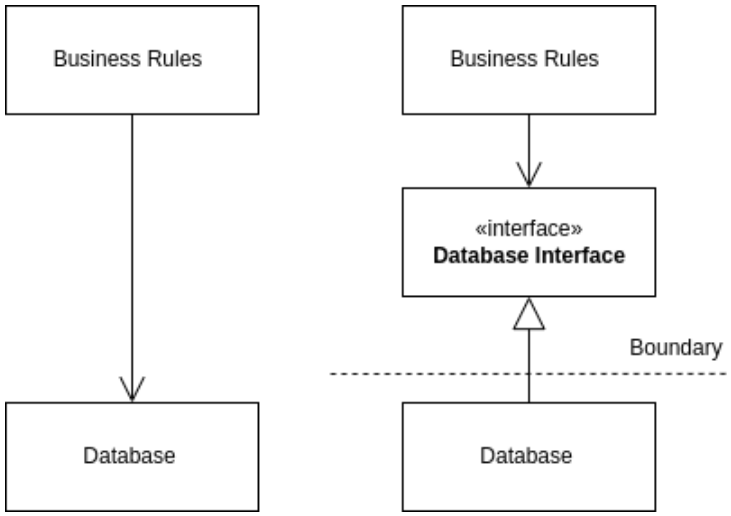


Figure 2.8: Setting boundaries

For example, let's say your business rules depend on a concrete class accessing the database. You want to draw a boundary between the business rules and the database. The first thing to do is to define a new interface representing the database in the business rules component. Then, have your business rules depend on the interface instead of the concrete database class. Then, have the concrete database class implement your interface. The wiring between the database interface and implementation will be done in your main method manually or by some dependency injection framework.

In this way, we made the database a plug-in of our business rules. That is, we can change our database, a low-level component, without affecting our business rules, a high-level component. Uncle Bob believes all low-level components should be plug-ins, for example, the GUI, I/O, web frameworks, etc. This leads to the concept of plug-in architecture.

2.5.4 Separating layers

Once we know how to separate component by setting boundaries, we can separate the components by layers. Layers are concentric and represent how fundamental the components are.

Source-level Dependencies should be organized According to the
dependency rule $\Longrightarrow$ Outer layers should depend on inner layers and not vice versa. we can remove violations of the dependency rule by setting boundaries between the layers.

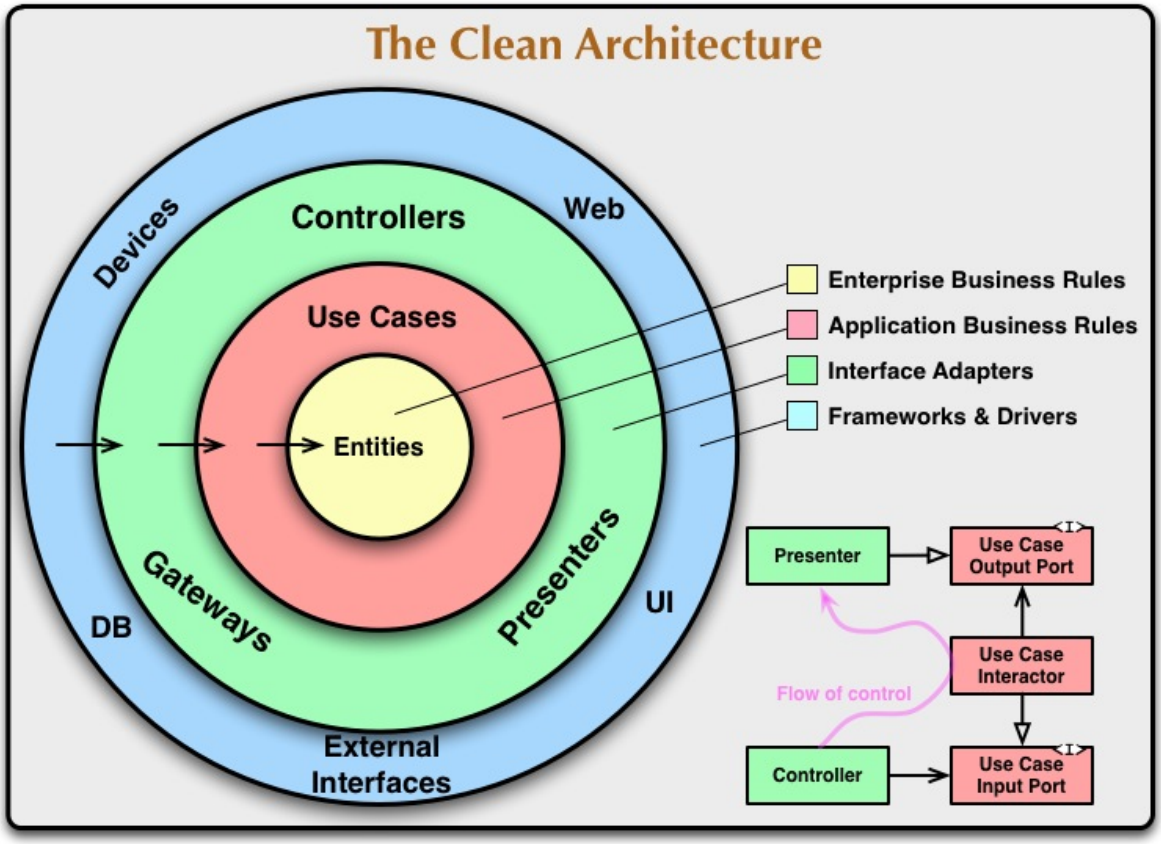


Figure 2.9: The Clean Architecture Rings

Image credits : Uncle Bob [11]

In the diagram above 2.9, we can identify four main layers, which are:

Domain Layer

At the center is the domain layer. It is the entities and data tha describe the subject area of the application, as well as the code to transform that data. The domain layer is the core that distinguishes one application from another.

This layer is the most important layer in the App. That if we mve from flutter to React, Or Angular to React, we won't have to change it.

The data structure of domain entities and essence of their transformation are independent from the outer world. External events trigger domain transformations, but do not determine how they will occur.

Entities encapsulate the data that describes the subject area of the application. It can be an object with methods, or it can be a set of data structures and function.

Use cases The software in this layer contains application specific business rules. It encapsulates and implements all of the use cases of the system.

Repository In this layer we only create an interface that defines the boundaries between the domain layer and the data layer.

Data Layer

Around the domain layer is the Data layer. This layer describes usecases [15], i.e. user scenarios They are responsible for what happens after some event occurs.

For example, the "Add to cart" scenario is a usecase. It describes the actions that are should be taken after the Button is clicked.

This layer consist of three main components: **Data source** This is the data source that is responsible for the data. It is the interface that the domain layer uses to access the data from the Outer world locally or remotely.

Data models When the data source is ready, it will call the domain layer to get the data. The domain layer will then use the data models to transform the data into the domain entities but the domain layer can't speak Outer language so in this layer we translate it.

repository This layer is responsible for implementing the interface that we already defined in the domain layer.

Presentation Layer

The presentation layer is the layer that is responsible for the user interface. And send the events that came from the user to deeper layers.

This layer is composed by two components (State management and the screens)

State management This layer is responsible for the state management of the application. It is responsible for the state of the application and the way it changes.

Screens This layer is responsible for the presentation of the application. It is responsible for the visual representation of the application. we can split this layer into many little widgets.

2.5.5 Advantages of the Clean Architecture

Now let's talk about the advantages of the Clean Architecture.

- **Separate domain**: All the main application functionality is isolated and collected in ne place

  functionality in the domain is independent, which means that it is easier to test.

- **independent Use cases**: Application scenarios, use cases are describes separately. This gives us more freedom t choose third party services. For example we can quickly change the payment system if the current one start bugging.

- **Replaceable Third-party Services** External services becomes replaceable because of adapters. As long we don't change the interface.

2.5.6 Costs of the Clean Architecture

Architecture is first of all a tool. Like any tool, the clean architecture has its costs besides its benefits.

- **Takes Time** The main cost is time. It is required not only for design, but also for implementation. because it is always easier to cal a third-party service directly than to write adapters.
  It is also difficult to think through the interaction of all modules of the system in advance, because we may not know all the requirements and constraints beforehand: when designing we need to keep in mind how the system can change and leave room for expansion.

- **Sometimes Overly verbose** In general, using clean architecture is not always convenient, and sometimes even harmful. if the project is small, a full implementation of the clean architecture is overkill.

- **Can Increase the Amount of Code** And these is a real problem for front-end because of the code that we give to the browser. more time to download and interpret.

Finally we can summarize the clean code book in these points: **General rules**

- Follow standard conventions.
- Keep it simple stupid. Simpler is always better. Reduce complexity as much as possible.
- Leave the campground cleaner than you found it.
- Always look for the root cause of a problem.

And make sure that you follow these rules in your code.

Names rules

- Use meaningful names.
- Use searchable names.
- Avoid encodings. Don't append prefixes or type information.

Functions rules

- Small.
- Do one thing.
- Prefer fewer arguments.
- Have no side effects.

Comments rules

- Always try to explain yourself in code.
- Don't add obvious noise.

When The **Code smells** means the code is not clean.

- Rigidity. The software is difficult to change. A small change causes a cascade of subsequent changes.
- Fragility. The software breaks in many places due to a single change.

- Immobility. You cannot reuse parts of the code in other projects because of involved risks and high effort.
- Needless Complexity.
- Needless Repetition.
- Opacity. The code is hard to understand.

2.6 Conclusion:

In this Chapter, we have seen the abstraction of the functionalities of our platform in the form of UML diagrams, And clean code By uncle Bob and his rules to be a better programmer. And we see how clean architecture works and interact, which will be the basis of the next steps.

CHAPTER 3

IMPLEMENTATION

Introduction

In this chapter of this theses, we will see how we implement our application. Starting with clean architecture Implementation, and what we choose as state management solution "Bloc".
Then what packages we use in our application. Then we will see main screens in the app.

3.1 Environment and development tools

Introduction

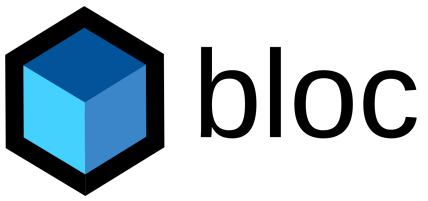
In this section will introduce to the tools that we used in the development process.

3.1.1 Frameworks And Libraries

Flutter is an open-source UI (User Interface) software development kit created by Google. It is used to develop applications for Android, iOS, Linux, Mac, Windows, and the web from a single codebase. [1]

Firebase Firebase is a platform developed by Google for creating online services as authentication or online database and mush products you can see further here [7]

Bloc State management (is information that can change the UI from build methods UI = f(State)) solution is a hot topic in flutter, we use bloc like our main state management solution. Bloc is a package from a family of stream/observable based patterns.
we choose it to make our application more events driven.[18]



gRPC (Remote Procedure Calls) was initially created by Google.It uses HTTP/2 for transport, Protocol Buffers as the interface description language, and provides features such as authentication, bidirectional streaming and flow control. Most common usage scenarios include connecting services in a microservice's style architecture, or connecting mobile device clients to backend services [8]

Get_it This is a simple Service Locator for Dart and Flutter projects with some additional goodies highly inspired by Splat [16]. It can be used instead of InheritedWidget or Provider to access objects e.g. from your UI.
we use it for dependency injection (when Separating UI from Logic we need a way to access these objects from the UI).[17]

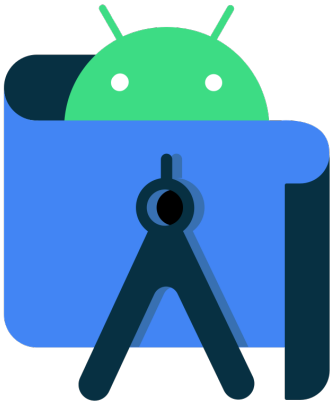
3.1.2 Languages

Dart is a client-optimized programming language for apps on multiple platforms. It is developed by Google and is used to build mobile, desktop, server, and web applications.[2]

3.1.3 IDE's

Visual Studio Code Visual Studio Code is a free source-code editor made by Microsoft for Windows, Linux and macOS. Features include support for debugging, syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded Git. Users can change the theme, keyboard shortcuts, preferences, and install extensions that add additional functionality.[4]

Android Studio Android Studio is the official integrated development environment (IDE) for Google's Android operating system, built on JetBrains' IntelliJ IDEA software and designed specifically for Android development.Its required for flutter developers to handels android SDK.[3]



3.2 Clean Architecture Implementation

To start with, we all know that working solo on a project will be an easy and smooth experience. However, it is not always the case in the industry. One of the essential skills for a software Engineer is to be able to work on large projects and collaborate with a team of people.

And some of the main areas in software engineering that will make collaboration much easier are code **maintainability** and **readability**. Basically, if the code is written in clean and systematical approach, then fortunately you can add new features on top of it without suffering.

for Front-end apps, we need to handle three things mainly. which are:

- User Interface
- Business Logic
- fetch Data

Three-tier Architecture is the most common architecture in the industry of Front-end apps.

The main idea of this architecture style, is to implement the three layers of the application.Which means each layer is responsible for a specific task.

For example, the "**presentation**" layer is responsible for handling the user interaction with the application only.

Another example is if you want to change something in your application APIs you'll go to the "**data**" layer.

3.2.1 Layers Interactions

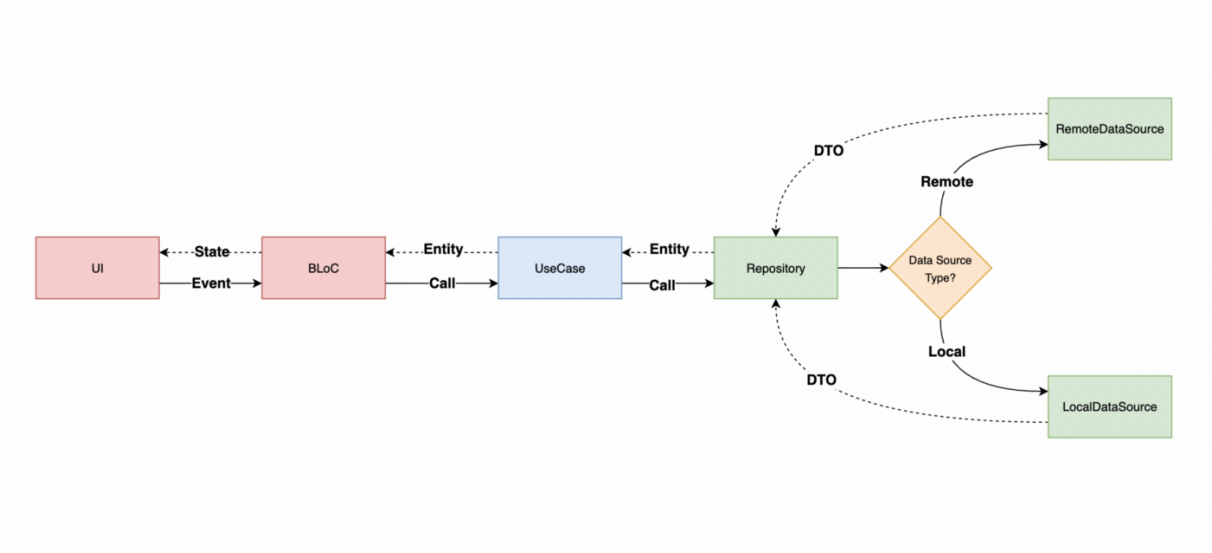


Figure 3.1: *Presentation* ⇔ *Domain* ⇔ *Data*

Before we explain each layer separately, we first need to fully understand the interactions between them.

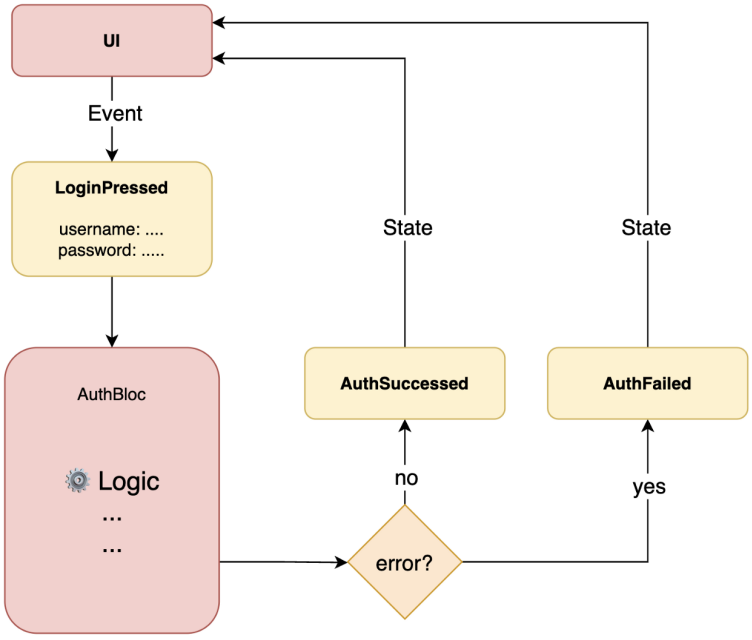


Figure 3.2: Simple Auth Flow

Bloc is the cornerstone of this architecture style, it will basically take "**Events**" from the user in the UI, then perform some logic, and based on the result of the logic performed it will emit a new "**State**". afterwards, we are going to display UI based on the new state value.

By applying **Bloc**, we can describe our system to be an Event-Driven one. And as a result, we can easily have a high level design of a feature, since it just an interaction of **Events** and emitting new **States**. Since now we have a brief on how **Bloc** state management works, it is time to describe the interactions between the layers.

In the beginning, the event will be fired from the UI by the user, for example the user presses the Login Button. That **Event** will be sent to the Bloc.

The Bloc will receive the **Event** and then call the **usecase** to perform whatever business logic needed. And here one point to mention, the usecase will be injected in the bloc, we can access it directly from the Bloc itself.

After reaching this point, we can have two cases. If we don't need to get data either remotely or locally, then the logic will be performed in the usecase and the value will be returned to the Bloc.

when the Bloc receives the keep in mind that sometimes we might use both **local** and **remote** data sources in the same repository for caching purposes.

Once the data is fetched from the data sources, it will be sent back as **DTO(Data transfer object)** to the repository. The repository will be responsible to convert the **DTO** to **entity** and then send it back to the usecase. Afterwards, the usecase can apply all the logic needed in this step. Next, we will use the returned value from the usecase inside the bloc.

Finally, the bloc will emit a new state to the UI.

3.2.2 Folder Structure

As our app scales and more features begin added, the project's folders structure might be a nightmare if not managed properly. So let's take a look at the folder structure of our app. When building an app from scratch, mostly we will have common buttons to be user all over the place, Text-Fields, and many other reusable widgets.

```dart
class TextBladi extends StatelessWidget {
  final String text;
  final double fontSize;
  final Color color;
  final FontWeight fontWeight;
  final TextOverflow textOverflow;
  final int maxLine;
  final TextAlign textAlign;

  const TextBladi({
    key,
    required this.text,
    this.fontSize = 18,
    this.color = Colors.black,
    this.fontWeight = FontWeight.normal,
    this.textOverflow = TextOverflow.visible,
    this.maxLine = 1,
    this.textAlign = TextAlign.left,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Text(text,
        overflow: textOverflow,
        maxLines: maxLine,
        textAlign: textAlign,
        style: GoogleFonts.getFont('Montserrat',
            fontSize: fontSize, color: color, fontWeight: fontWeight));
  }
}
```

Figure 3.3: Text user all over the app

However, sometimes we design a widget specifically for one feature only.

Note that utils classes, colors,theming, and many more are also considered to be common.

In addition, we will have a common presentation layer, where we gonna save the common, widgets, dialogs, and pop-ups....

Also, a common domaine layer where we will have common entities and usecase classes.

Lastly, a common data layer to fetch data commonly used across the app.

And all those common files will be saved in a folder called "**core**".

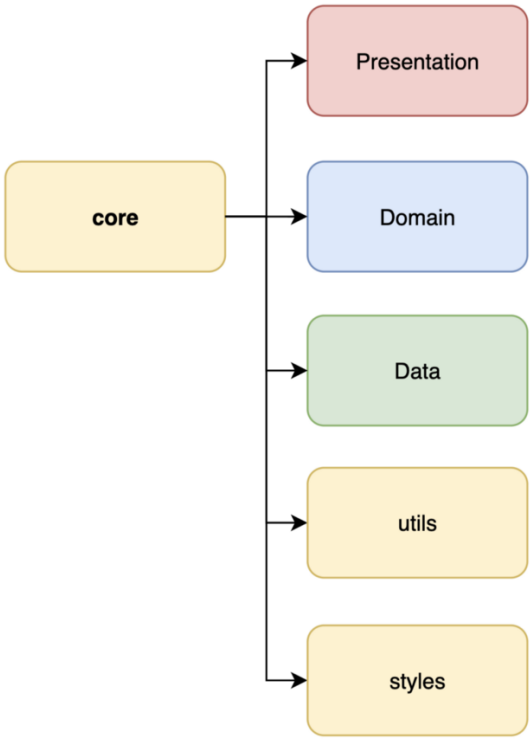


Figure 3.4: Core Folder

Sometimes we add more folders, for example validators, services and other common classes. The main idea is just to group the commonly used files in one folder .

The second main folder will be called "**features**".

Each major feature in the app will have a separate folder with its own "**presentation**", "**domain**", and "**data**" layers.

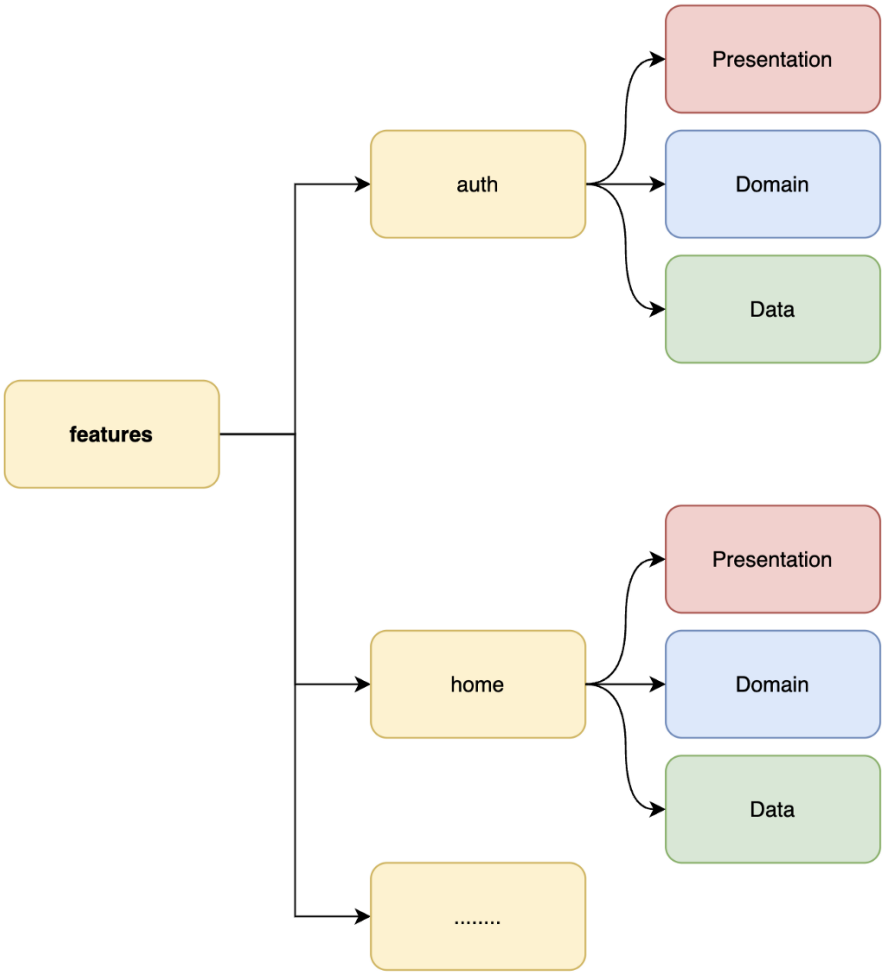


Figure 3.5: features Folder

The beatty of this structure, is that you will not be lost when you want to modify existed code.

For example, you want to change something in the UI of the homepage. Then you'll go directly to "features" folder, and since we are talking about homepage then you'll go to "home" folder. And finally you'll open "presentation" folder since what you want to change is the UI.

3.2.3 Presentation Layer

From the name itself, the presentation layer is responsible for the user interaction and see on the screen. Mostly in this layer, we are going to have the screen itself and the widgets used inside it, and also the Bloc used for thar screen.

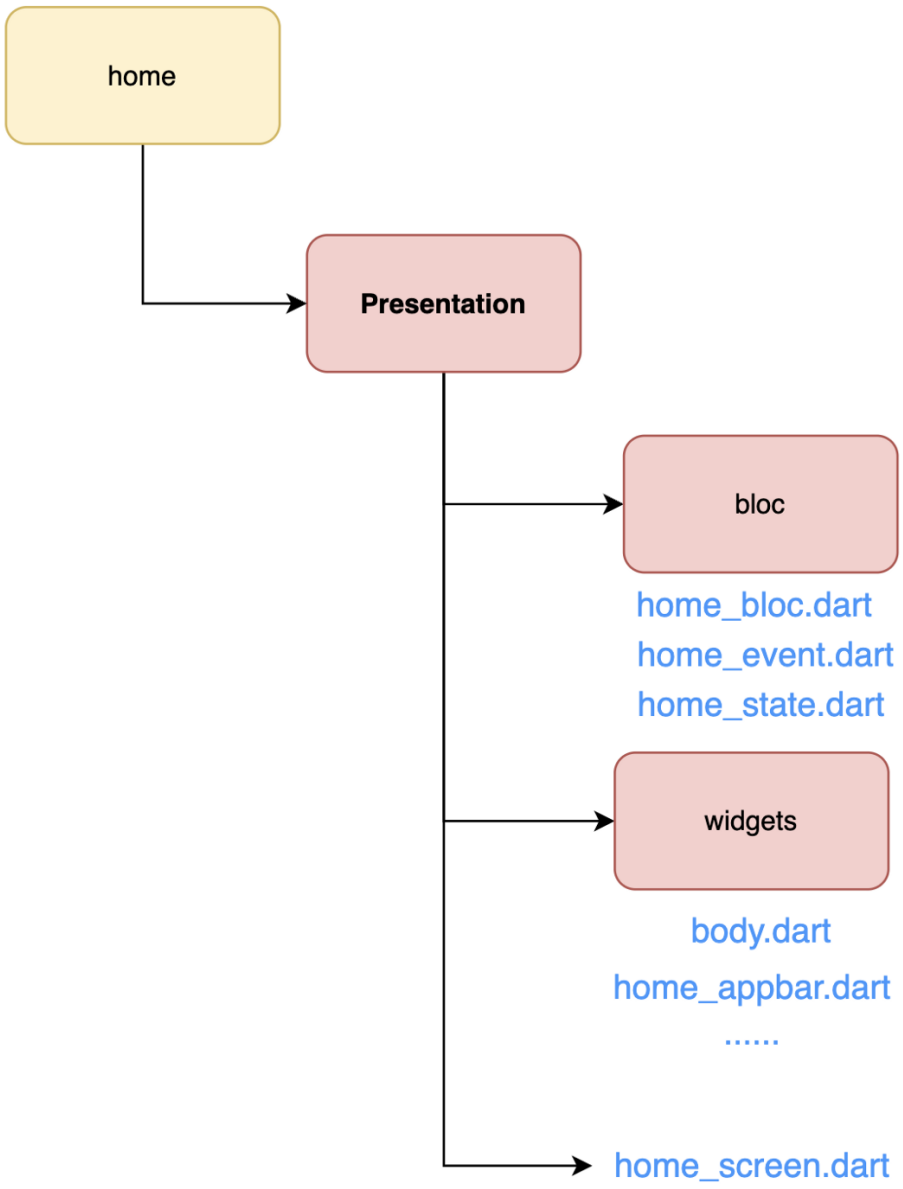
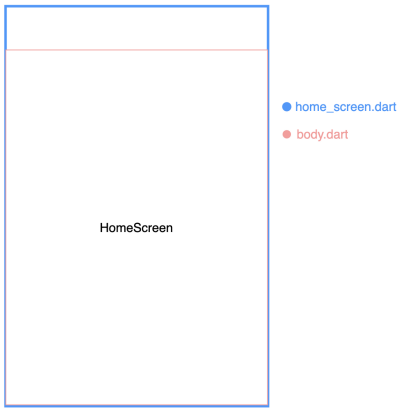


Figure 3.6: Example for home feature presentation layer

As we noticed, we have three elements inside the presentation layer.
Let's take a look at the first one, which is the screen itself.

homeScreen.dart: this file will be the placeholder and the scaffold for the home screen, and for its body we will use "body.dart". In this file, also we will provide the AppBar if needed, and also provide the BLoC to the screen.



(a) Diagram for home feature UI

```dart
class HomeScreen extends StatelessWidget {
  const HomeScreen({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return BlocProvider(
      create: (context) => sl<AuthBloc>(),
      child: const Scaffold(
        appBar: AppBarBladi(),
        body: Body(),
      ),
    );
  }
}

class Body extends StatelessWidget {
  const Body({
    Key? key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return const Center(
      child: Text("HomeScreen"),
    );
  }
}
```

(b) "homeScreen.dart" code

Bloc: inside it, we will create the **Bloc** that will handle all the logic and state changes inside the home screen.

Widgets: this folder will contain all the widgets used specifically for home screen.

3.2.4 Domain Layer

Wrapping all the app's business logic, will give us the freedom to easily change the state management solutions.

Because if we write all the logic inside the **Bloc**, then it will will be hard to change to another state management solution.

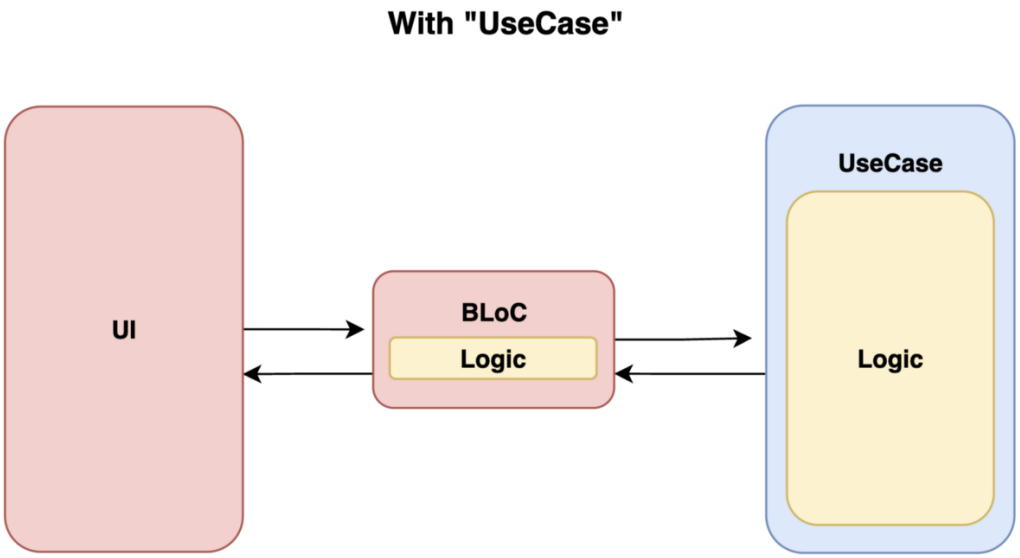


Figure 3.8: BLoC and usecase communication

For sure we still going to have a business logic written in the Bloc, but all the heavy logic will be performed in the usecase instead.

Now whenever we want to change the state management solution, we will just change the small layer in between. we won't have to re-write the heavy part of the logic again.

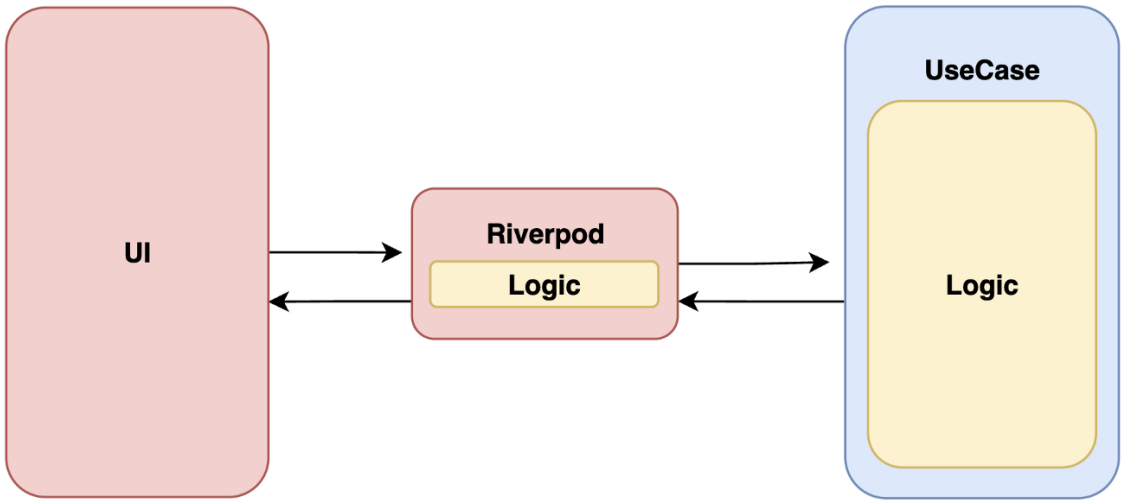


Figure 3.9: change the state management solution with Riverpod Or Getx

If we don't depend on usecase to handel the heavy part of the logic, then it will be challenging to change the state management solution.

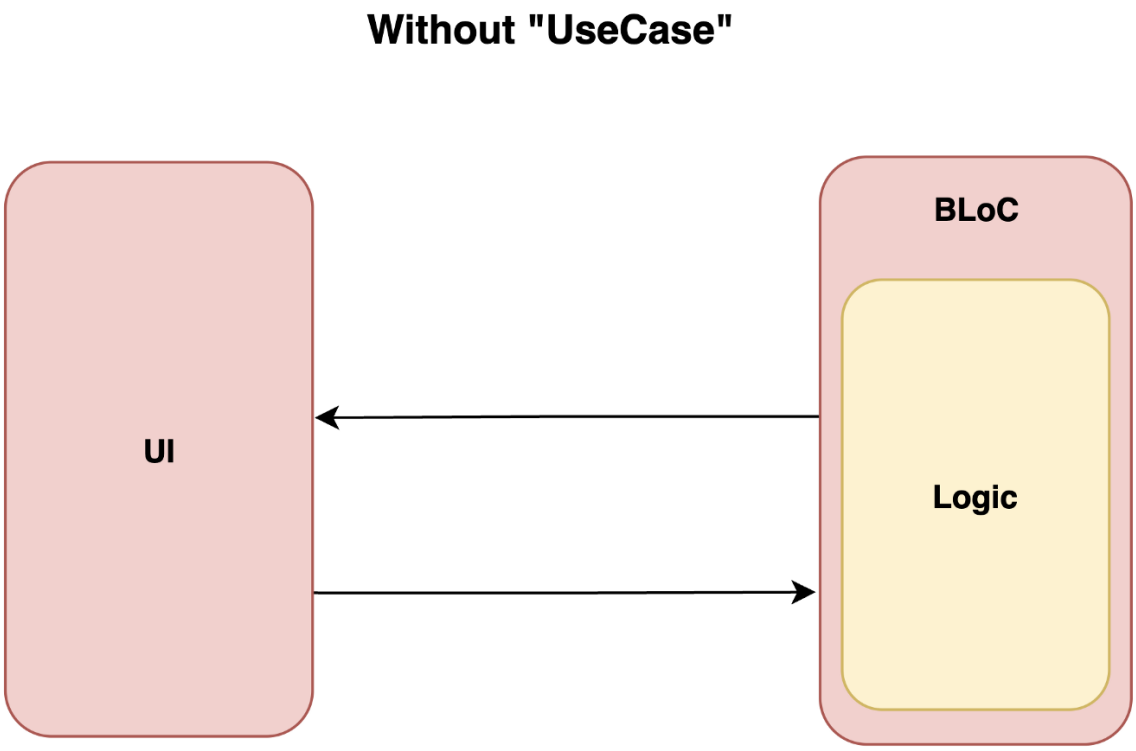


Figure 3.10: All the logic is in the Bloc

The other component we will have in the domain layer is the "**entity**".
An entity is capsulation of whatever data needed to represented in the UI.
For example, if we have a list of products, then we will have a list of products entity.
Also, it will be easier to share entities between widgets than sharing a json object.

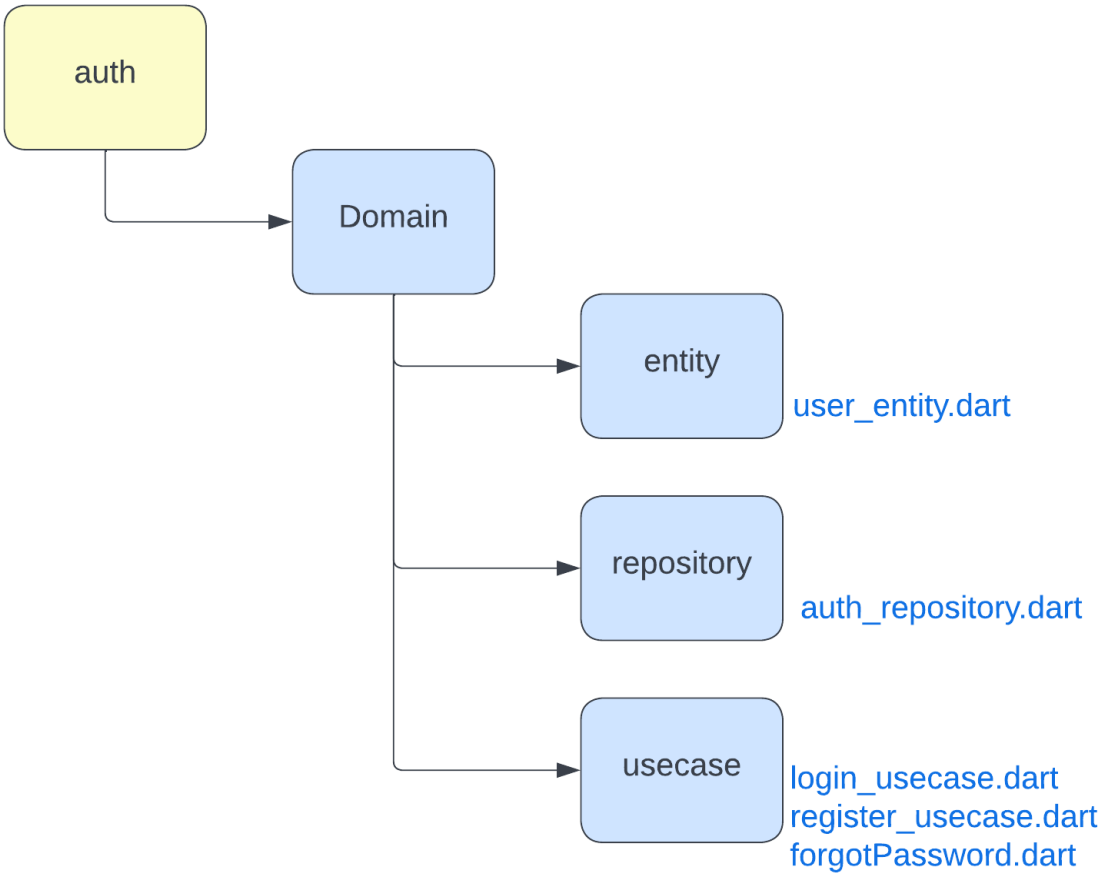


Figure 3.11: Example for auth feature domain layer

3.2.5 Data Layer

Now talking about the last layer, which is sometimes called the "**infrastructure**" layer. In this layer we have three main components, which are **repository**, **dataSource**, and **DTOs**.

Let's start from the lowest level, which is the **data source**.

Data source's main job is to fetch data from specific destination.

Some times we need to fetch it externally via http requests, in that case it will be a "remote data source". And some times we need to fetch it from the local database, in that case it will be a "local data source".

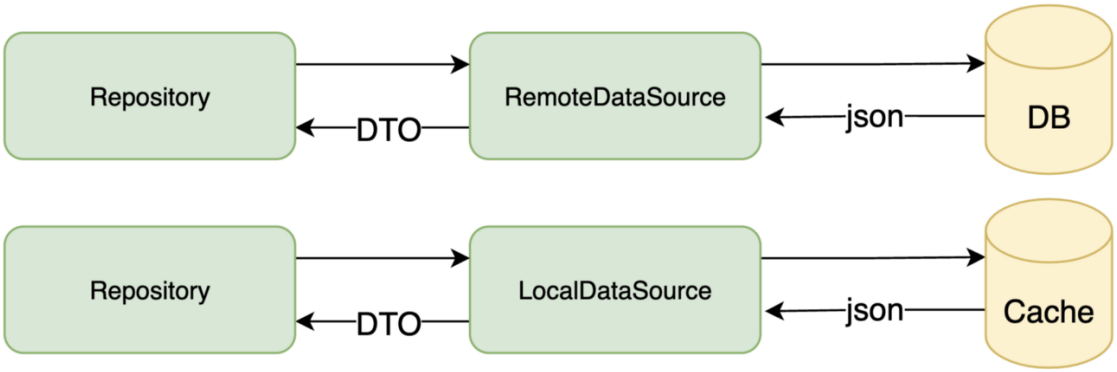


Figure 3.12: DataSources fetching data

As we can notice from the diagram, the data that will be returned to the **repository** will be in the form of a **DTO**.

we know that when we fetch data from the network, then mostly we will receive the response in the form of a json object, and dealing with json files directly might cause errors.

The issue happens usually when we try to access the data inside that json object by keys manually. Imagine inserting the wrong key inside th json by mistake.
DTO will solve this problem by parsing the data directly from the response into an object.

After creating **DTO**, we don't have to access the data by using keys. we will be dealing with solid objects and its variables.

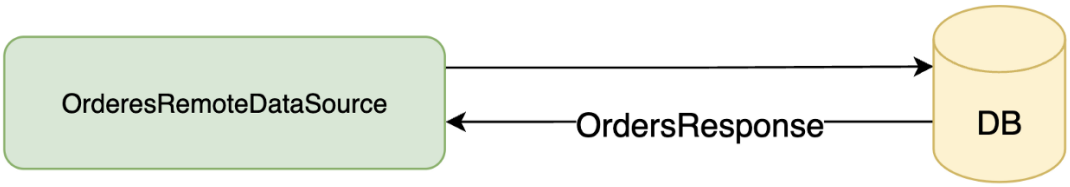


Figure 3.13: Example of DTO → OrderModel Object

repository will be the implementation of the repository in domain layer and it will be the wrapper of the data sources needed to achieve the business logic.
By applying this, the usecase will be dependent only on one file even if the data is being fetched from multiple sources.

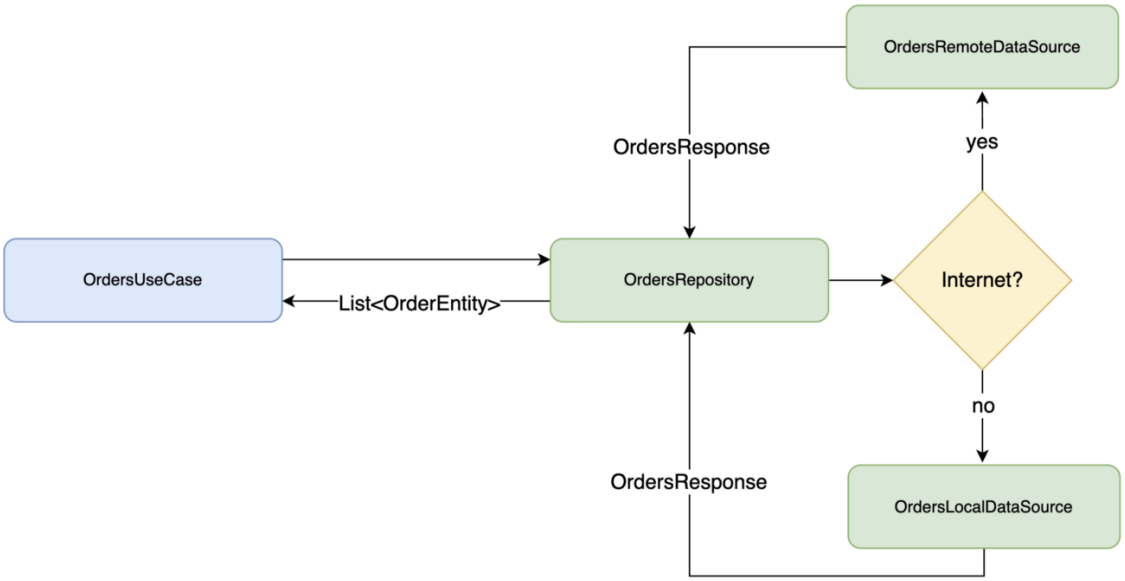


Figure 3.14: Example of a repository with multiple data sources

Basically, the usecase want to have a list of order entities to give it back to the Bloc so it can displayed on the UI, and it does not care if the data is being fetched locally or remotely. No we can have this logic inside the repository, will check for internet connection. If we have it, we call the remote data source Otherwise local data source will be called.

Which means, the Bloc and the usecase does not have to know about how the data is being fetched explicitly.

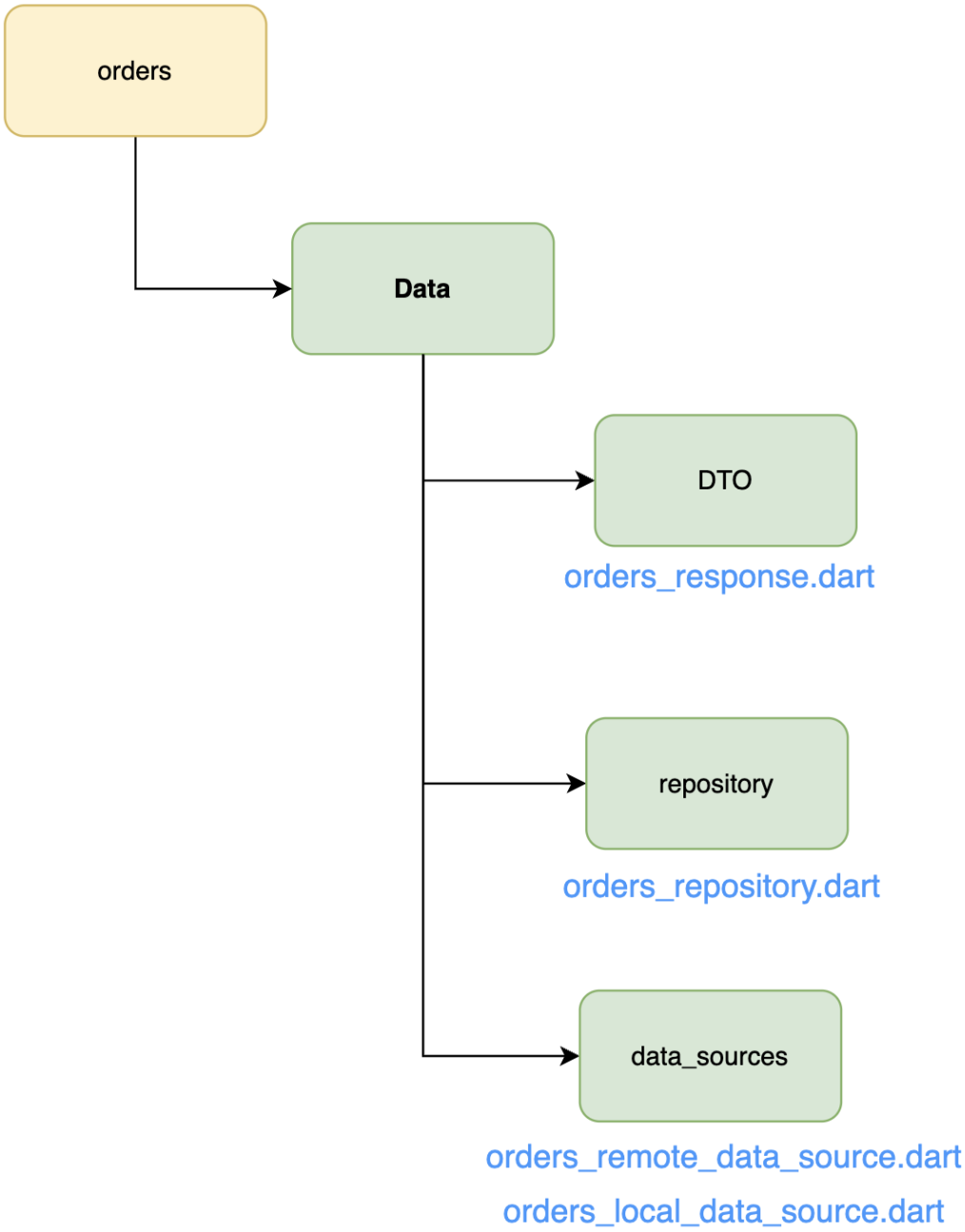


Figure 3.15: Example for orders feature data layer

3.3 Main Screens

Introduction

When we design the app, we make sure that we have a clear and simple UI.
We also want to make sure that the user can easily understand the app and navigate through it.
so we create this clean design starting with the not authorized user interface. then he can easily navigate to the homeScreen of each role like admin of store, regular user, and delivery guy.

3.3.1 Not Authorized User Interface

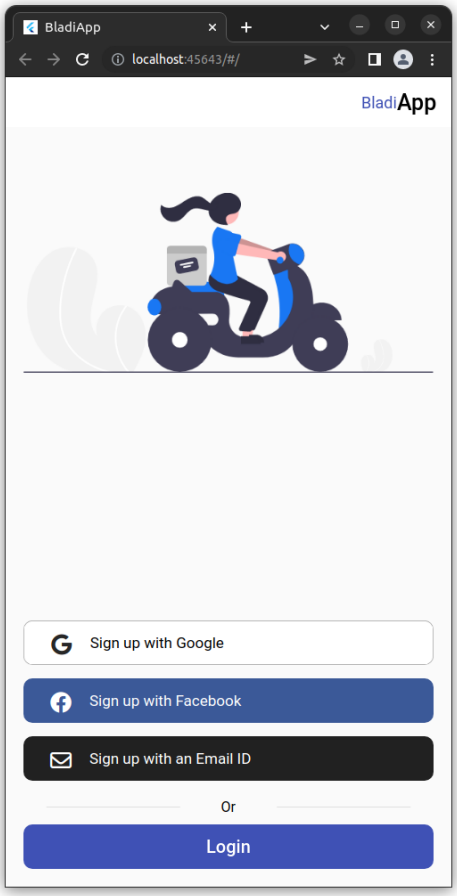


Figure 3.16: Introduction Screen

when the user is not authorized, we will show him the login screen by default. If he wants to register he can navigate to the register screen.
the user also can sign in by using his credentials in Facebook or Google.

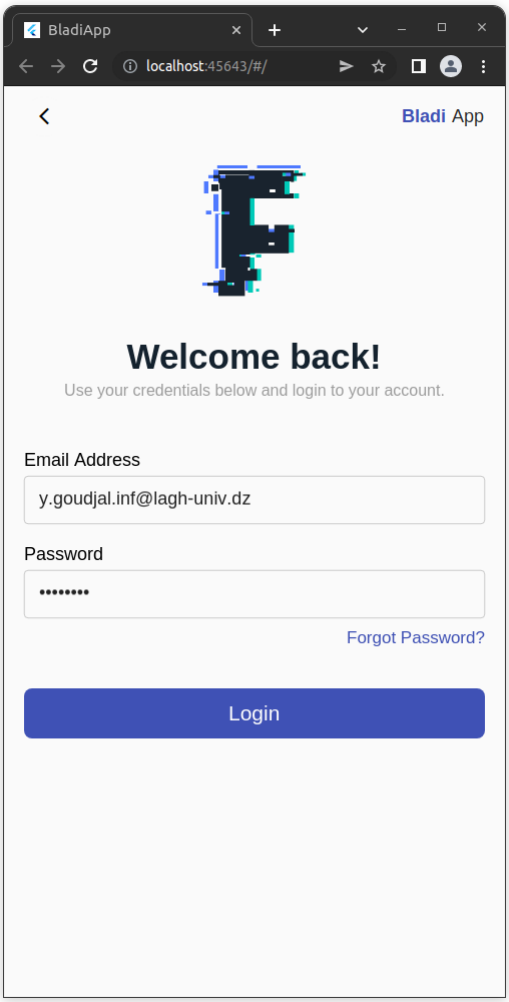


Figure 3.17: Login Screen

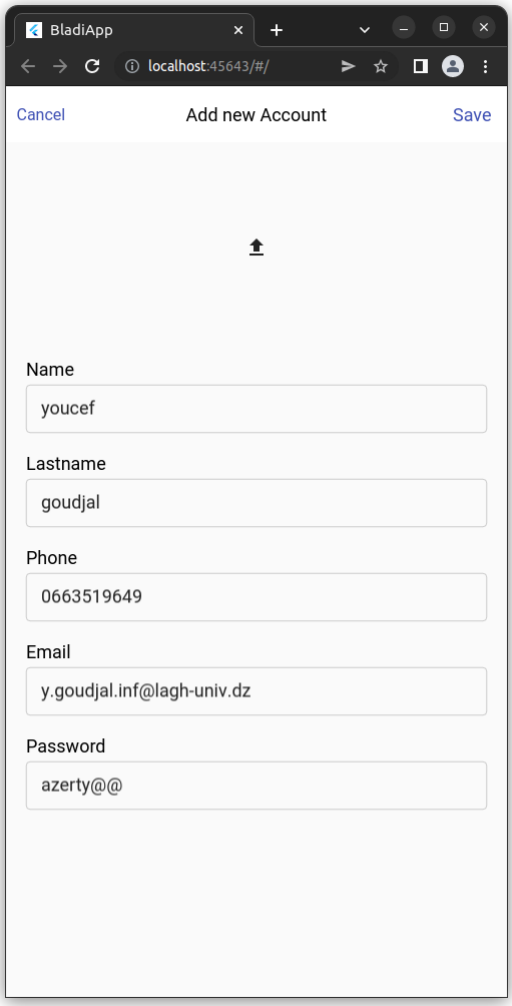


Figure 3.18: Register Screen

3.3.2 Authorized User Interface

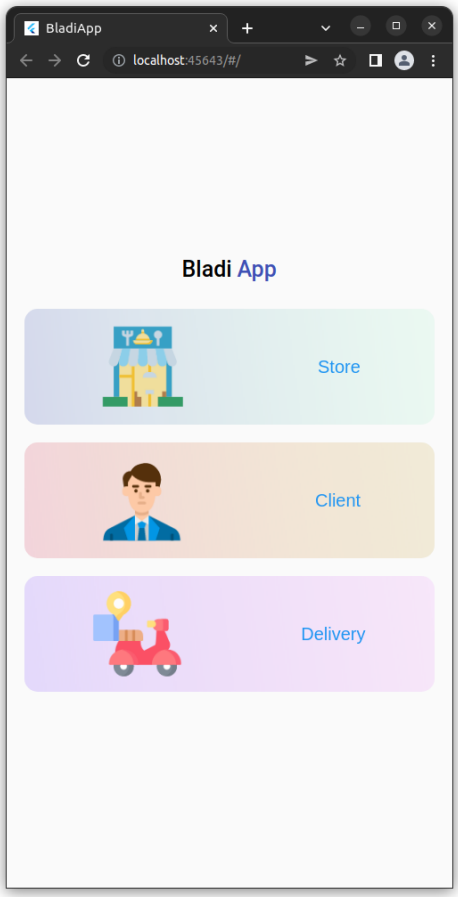


Figure 3.19: Home Screen

Now the user is authorized, he can navigate to the home screen of each role like admin of store, regular user, and delivery guy.

Store Screens

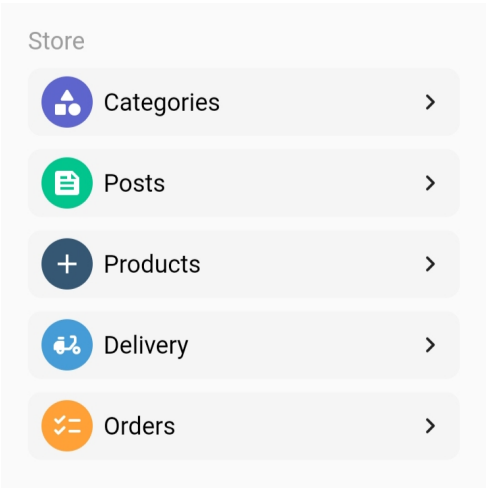


Figure 3.20: Store profile

The landing page of the store is profile screen with some options like categories, products, and orders...

Categories : In This screens the store admin can see the categories and can add new category.

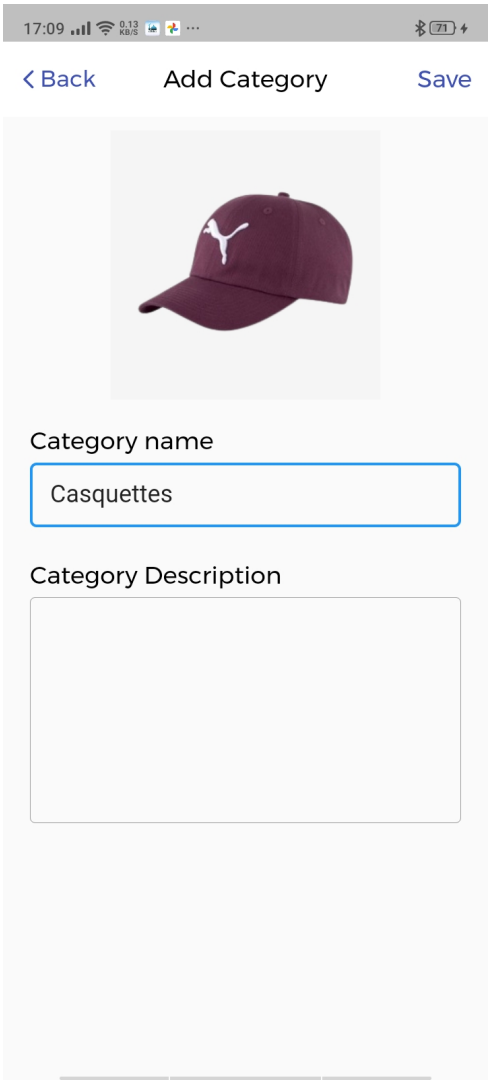


Figure 3.21: Add Category Screen

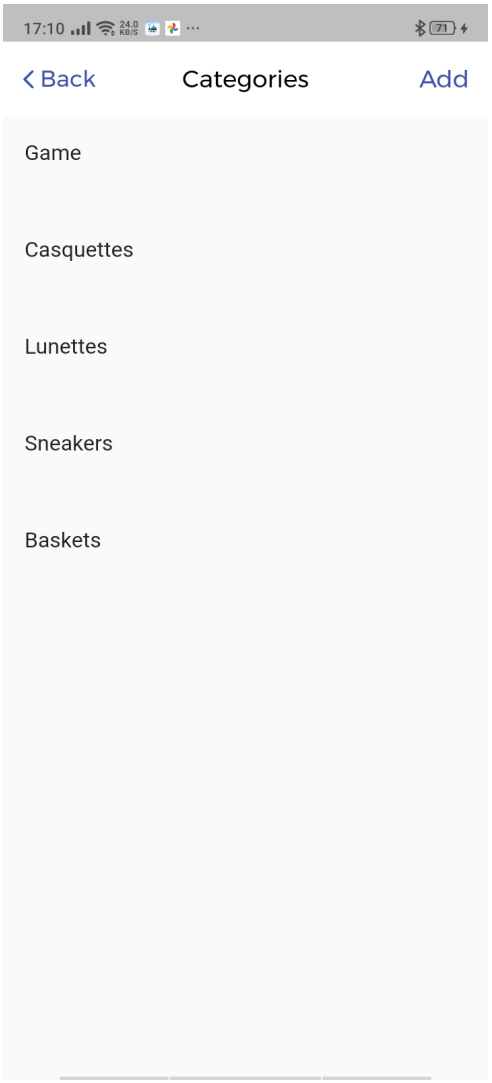


Figure 3.22: Categories Screen

Products : In This screens the store admin can see the All the Products and can add new product. By Long press on the product, he can edit or delete it or switch the availability of the product.

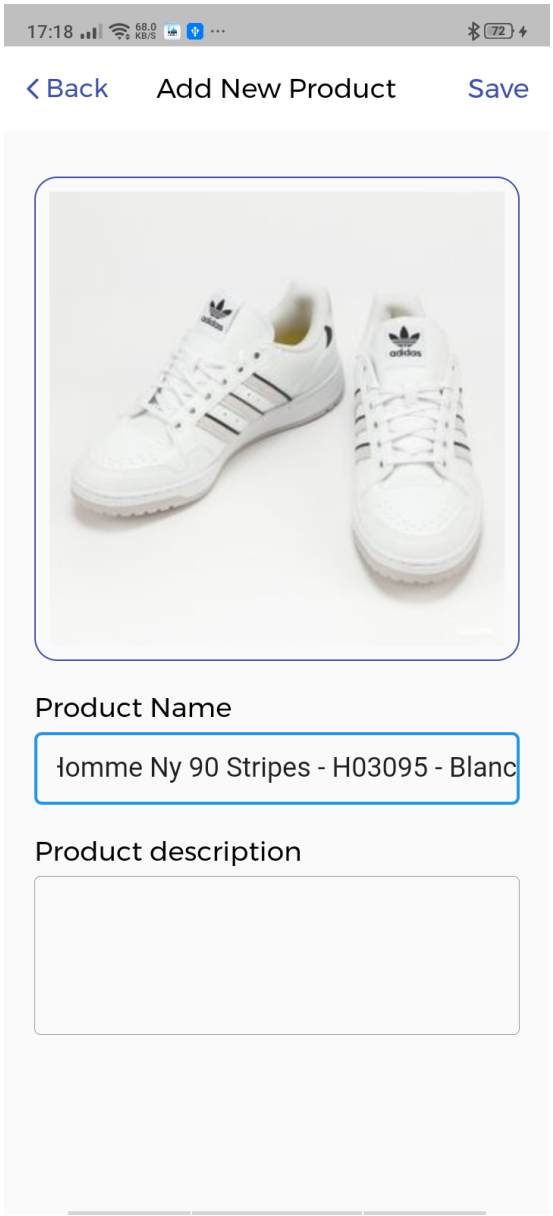


Figure 3.23: Add Product Screen

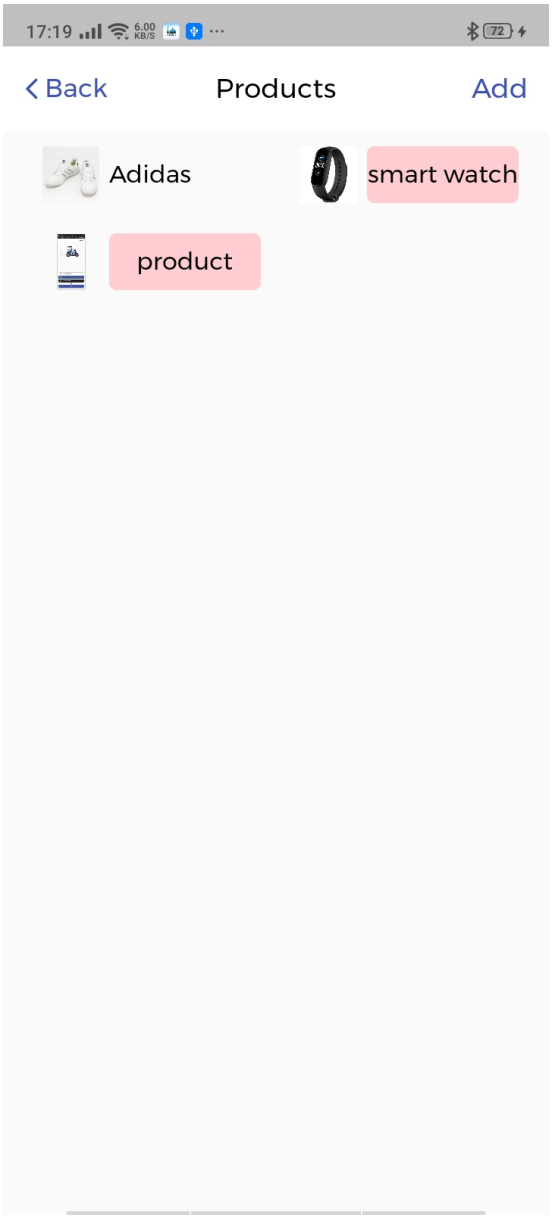


Figure 3.24: Products Screen

Regular User Screens

The regular user can see the products and purchase them. Or he can see posts he liked and comment on them. On the top of the screen he can see the cart and the total price of the products in the cart.
when he complete the purchase he can see the order and the total price of the products in the order.

In the home screen the client can navigate between his profile and posts and his bag.

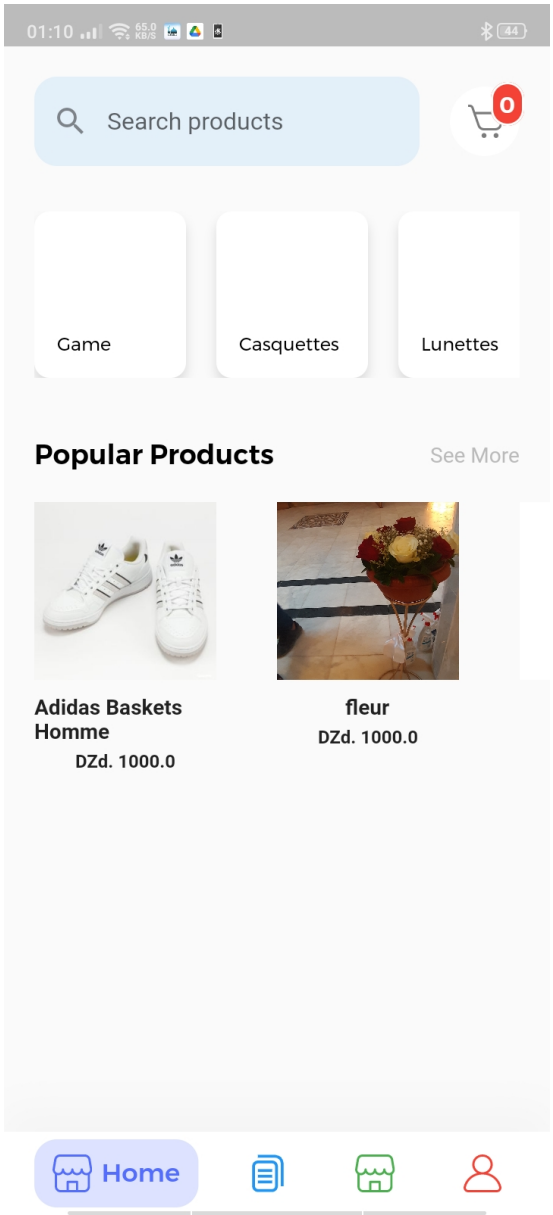


Figure 3.25: Client Home Screen

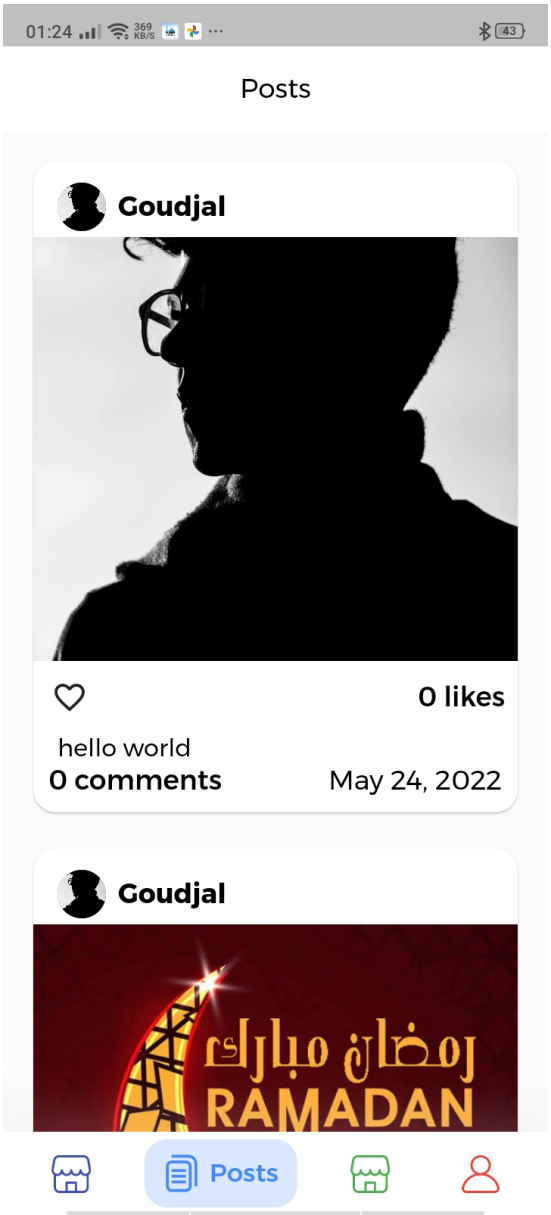


Figure 3.26: Client Posts Screen

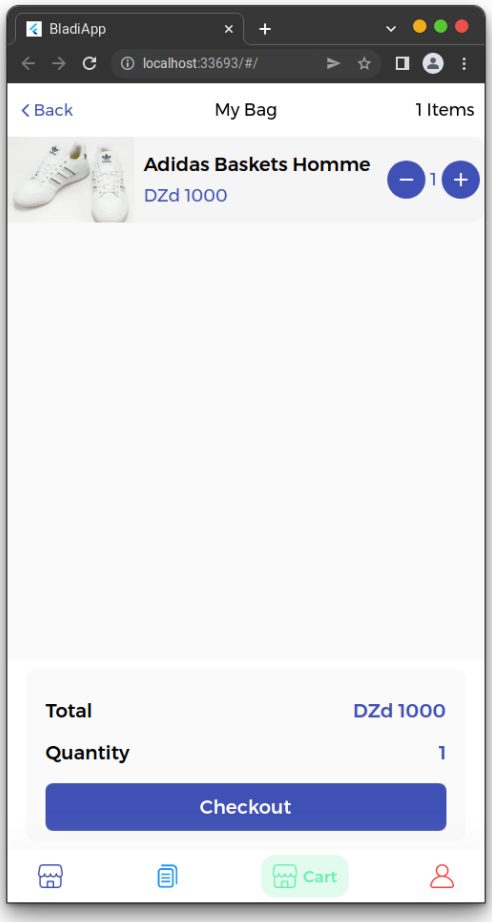


Figure 3.27: Client Bag Screen

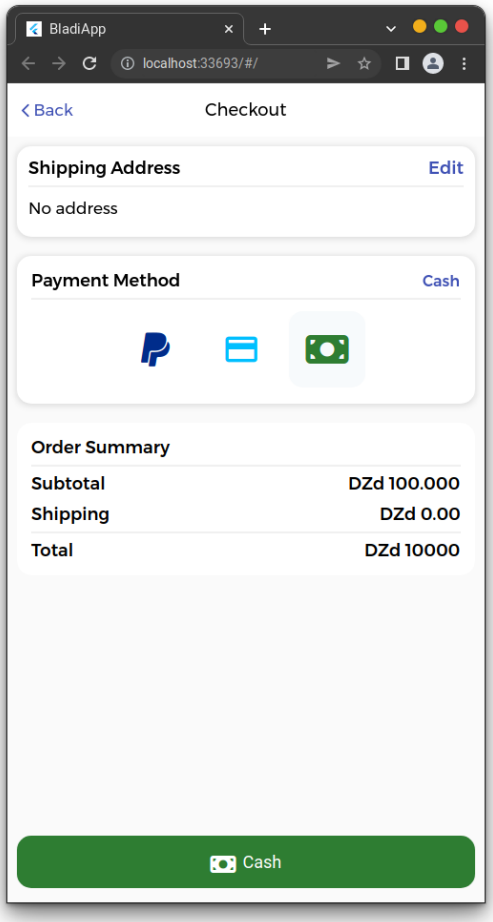


Figure 3.28: Confirm Order Screen

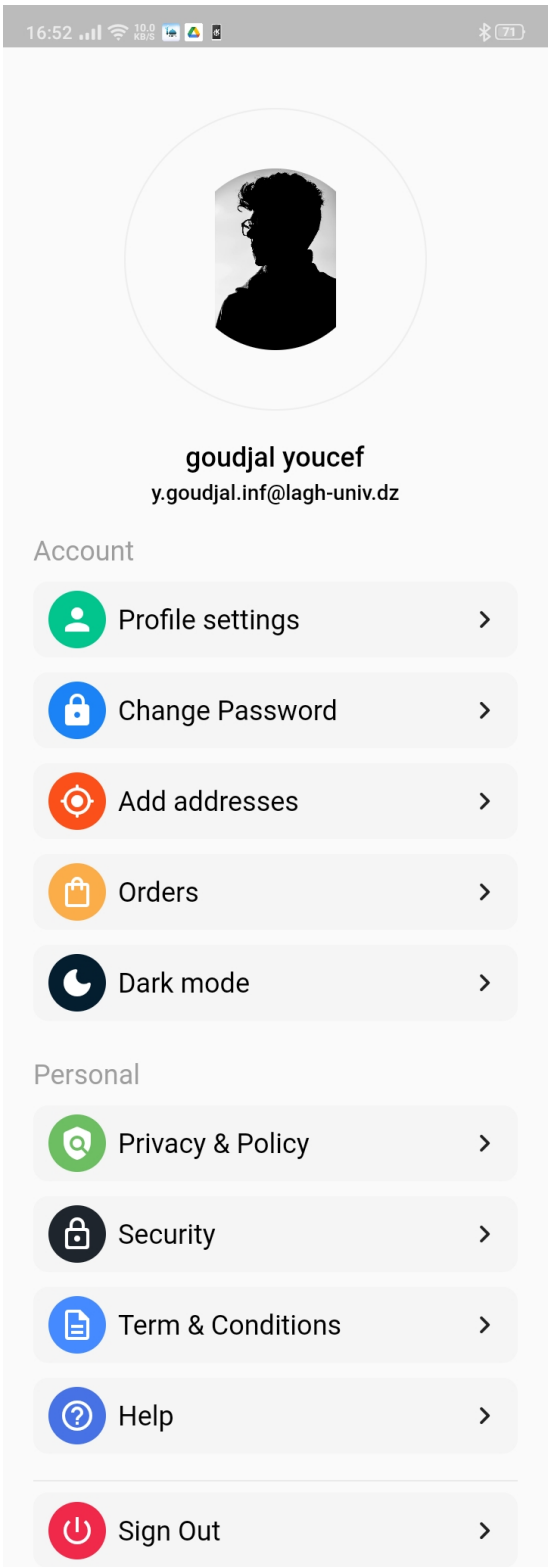


Figure 3.29: Client Profile Screen

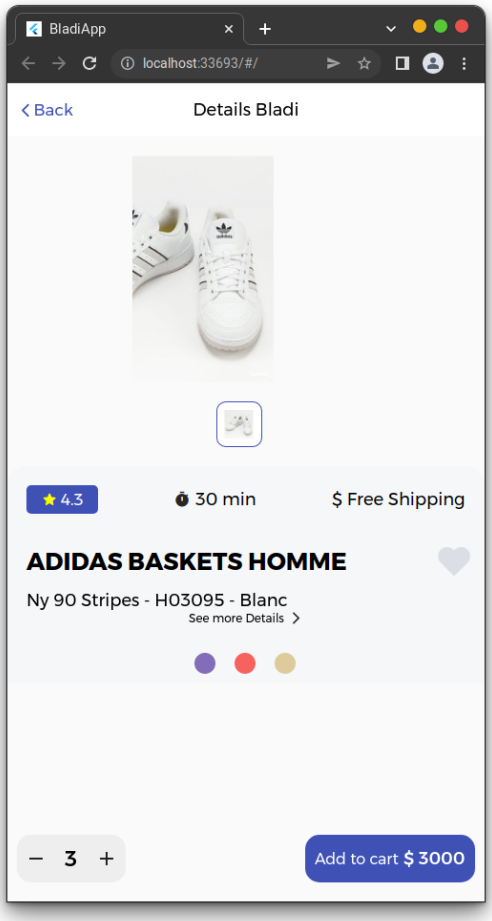


Figure 3.30: Products Details Screen

In Profile Screen the user can edit his profile image and name and can change his password.
also he can see his orders and the total price of the products in the order.
Finally he can logOut from the app.

Posts : In This screens the store admin can see the All the Posts with comment and likes count and can add new post Or delete one.

Figure 3.31: Add Post Screen

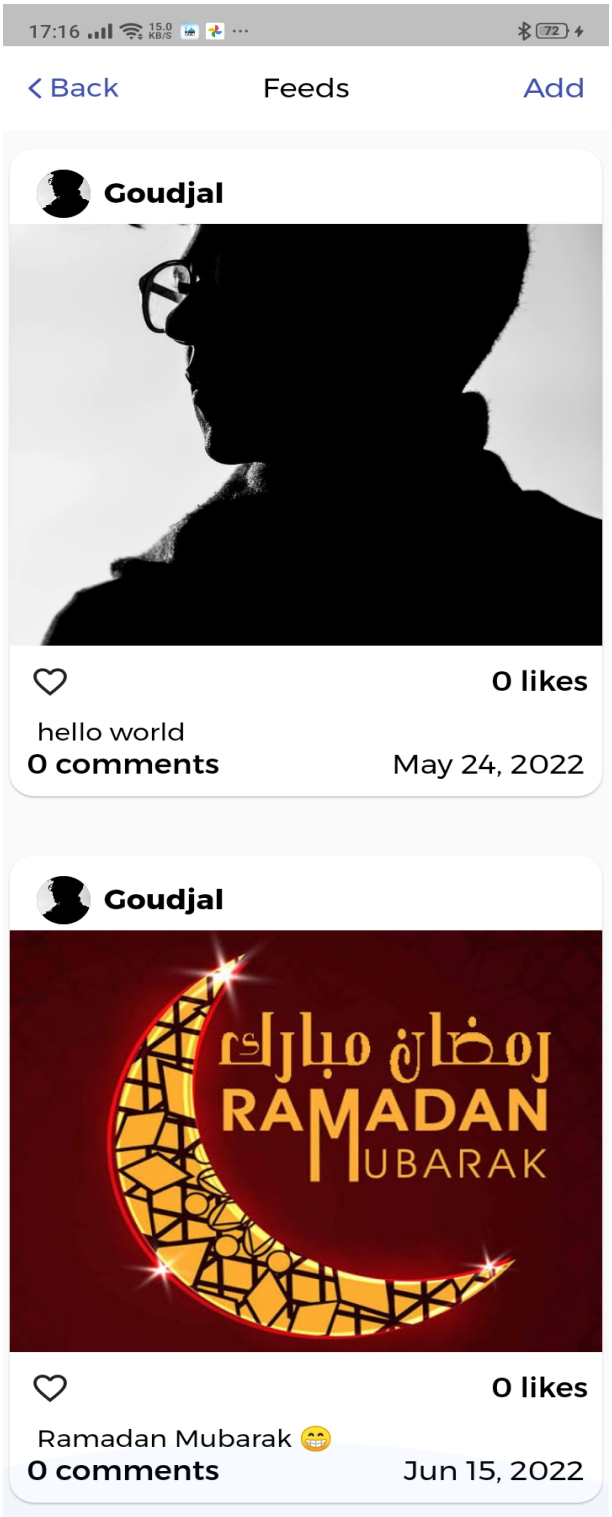


Figure 3.32: Posts Screen

3.4 General Conclusion

Our work was inspired by the lack of a real solution to the problem of local and national customers and sellers, our goal is for our platform to be Connecting potential customers and sellers to expand e-commerce Available and convenient. In the first chapter we talked about e-commerce in We gave a historical overview of it, then emphasized the importance of electronic commerce and its effects and repercussions Technological development in the field, then give us a look at some of what is related to it It works in addition to comparing them to each other from different aspects. in In the second chapter we define the architecture of our platforms through UML Diagrams that helped us decide how to implement the functionality of the system. Finally in Chapter Three, we have listed the devices used To develop and test this work, we detailed our options in Software and architecture. Then we provided screenshots for further clarification System functions in more detail

What we have learned

Through the duration of our project we have learned many new things that helped change and shape our perspective on software development:

- We learned a new programming language (Dart).
- We learned a new development architecture(Clean Architecture) that helped us better organize our code.
- We learned to use new programming tools (Grpc) that have facilitated the development of our platform.
- We learned how to conduct a proper research and organize our findings.
- Finally we learned how to write a proper thesis using latex.

BIBLIOGRAPHY

[1] Flutter documentation : Flutter https://flutter.dev/docs (Visited 23/06/2022).

[2] Dart : Dart programming language . **https://dart.dev/** (Visited 23/06/2022).

[3] The official site of Android studio. **https://developer.android.com/docs** (Visited 23/06/2022).

[4] Vs-code website Visual Studio .**https://visualstudio.microsoft.com/** (Visited 23/06/2022).

[5] UML .**https://www.uml.org/** (Visited 23/06/2022).

[6] Git.**https://git-scm.com/** (Visited 23/06/2022).

[7] Firebaseofficialwebsite. **https://firebase.google.com/** (Visited 23/06/2022).

[8] gRPC Home page .**https://grpc.io/** (Visited 23/06/2022).

[9] Flutter TDD Clean Architecture Course (Visited 23/06/2022).

[10] The Clean code Blog By Uncle Bob **https://resocoder.com/** (Visited 23/06/2022).

[11] Clean Code by Robert C. Martin **https://resocoder.com/** (Visited 23/06/2022).

[12] Clean Architecture, the right way By Angad Sharma medium story **https://medium.com/gdg-vit/clean-architecture-the-right-way-d83b81ecac6** (Visited 23/06/2022).

[13] Introduction to Clean Architecture By Bruno Joaquim medium story **https://medium.com** (Visited 23/06/2022).

[14] Clean Architecture By Anmol Sehgal medium story **https://anmolsehgal.medium.com** (Visited 23/06/2022).

[15] Use case wiki https://en.Wikipedia.org/wiki/Use-case (Visited 23/06/2022).

[16] splat GitHub https://GitHub.com/reactiveui/splat (Visited 23/06/2022).

[17] get_it package https://pub.dev/packages/get-it (Visited 23/06/2022).

[18] flutter Bloc package https://pub.dev/packages/flutter-bloc (Visited 23/06/2022).

[19] Uml Diagramm https://www.smartdraw.com (Visited 23/06/2022).

[20] Ouedkniss https://fr.wikipedia.org/wiki/Oued-Kniss (Visited 23/06/2022).

[21] uber Eats https://fr.wikipedia.org/wiki/Uber-Eats (Visited 23/06/2022).

[22] Jumia Food https://fr.wikipedia.org/wiki/Jumia (Visited 23/06/2022).