Software Architecture, Edinburgh Napier University

40401270@live.napier.ac.uk

Davide Pollicino

Appendix

1. Abstract

At the base of each robust and maintainable software applicative, there is specific software architecture suitable or designed in according to the applicative requirements and necessities.

Software system's architecture is defined as the set of principal design decision about the system, which are fundamentally based on four key elements

- Structure
- Behaviour (or requirements)
- Interactions between software components
- Non-functional requirements.

This report will present the software architecture analysis and adoption, design and implementation for the DE_Store management software application, a decentralised applicative used by managers of a warehouse to manage, and receive information about sales performances, and status of the stock within the warehouse.

In the section dedicated to the software design, project requirements, assumptions, user stories and Database ER model will be analysed.

2. Possible software architectures comparisons

In according to the project specification and company organization, DE_Store is a distributed system used to manage the stock management of multiple warehouses of a company.

Undirect requirement (also called non-functional), are the one which are considered needs fo the software applicative, without being explicitly listed as formal requirement, like: scalability, security, performances, availability.

A distributed System is composed by a set of independent components usually located on different machines which communicate between them via messages in order to achieve a common goal and complete a task. A distributed system is commonly defined by the following characterises and features:

- **Scalability:** ability to grow as the size of the workload increases.
- **Concurrency:** distributed systems components run simultaneously.
- **Availability:** when a node (an elaboration component of a distributed system) fails, others are able to self-balance the workload, keeping the system available (up) and reachable by any clients.

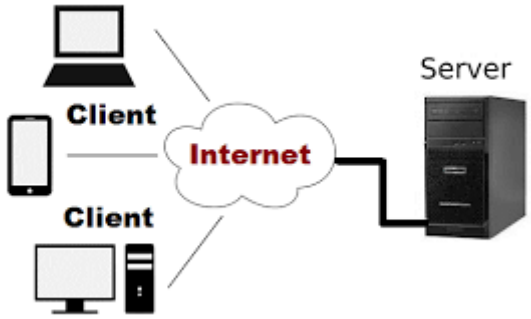
In a distributed system with need for high scalability, availability, and long-term maintenance, two are the most recommended software architectures that would satisfy our direct and undirect requirements:

- **Client server Architecture**
- **MVC: Model View Controller**

2.1 Client server Architecture

Client-Server architecture is a shared architecture system where loads of client-server are divided. The client-server architecture is a centralized resource system where server holds all the resources. Indeed, just the client is able to forward a request to the server, and then interpret any kind of server response.

In the client-server architecture, when the client computer sends a request for data to the server through the internet, the server accepts the requested, process it and deliver the data packets requested back to the client.
A resource given could be an information from a database, a web server or a printer (physical device).
One special feature is that the server computer has the potential to manage numerous clients at the same time. By an automatic provision of the hardware resources and guarantee so a **horizontal scalability**, the server will be able to manage an increasing number of **clients requests**, without degrading performances and being reachable by ay client.



Client-server Architecture: teachcomputerscience.com

Most of the most recent distributed system architecture instead, has evolved into an architecture similar three-tier architecture, called also: MVC, or simply, Model View Controller.

As the MVC framework is often confused with the 3 Layer Architecture, we will analyse below the key difference MVC and 3-tiers architecture, prior to indicate the preferred architecture to address all DE_Store current and future needs.

In the case of **three-tier software architecture,** we can rely on a specific layer, the logical layer (also called middle-tier), to do the processing and decision making. In the three-tier architecture, the application is splatted into three logical and physical computing tiers.

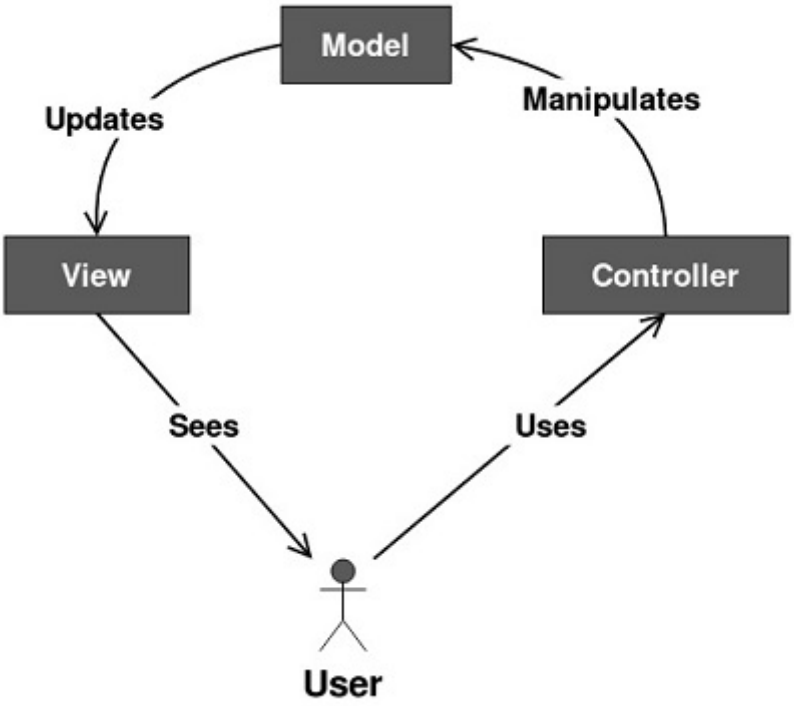
- **Middle-tier**: which is responsible for the logic part of the application.
- **Data layer:** which is responsible for any operation related to database management and data manipulation.
- **Presentation layer:** graphical layer, used by the client to interact with the middle-tier.

2.2 Model View Controller

MVC (Model-View-Controller) architecture instead, is an architecture which relies on these three entities:

- The **model:** what acts as interface to data. It is responsible for maintaining data and provides the data structure behind the entire application and is represented by a database (MySQL in our case).
- The **view:** or simply GUI: is what we would see in our browser to interact with the
- **Controller:** component which accepts input and convert it as commands for the model or view.
    - Its response to user input and execute related operations on the data model.
    - Validates the user input before to do update or write operations on the data model

As we have can see, in the MVC software architecture, the user interface divides the related program logic into the three interconnected elements.



2.4 Ideal Software Architecture

A fundamental **difference between the three-layer architecture and MVC** is that in 3-tier architecture, the client tier never communication directly with the data tier. Instead, all the communication must pass through the middle tier (logic layer). However, the MVC architecture has instead a triangular relationship between tiers. The view will send updates to the controller, the controller will interact worth the model and the view will get updated directly from the model.

When the MVC pattern is used a larger portion of the UI code can be united tests, or even better, being tested using **end-to-end tests,** which will replicate the behaviour of the user

Following the project goals, expectation, the MVC software architecture has been found for the reasons listed above the best suitable software architecture for the implementation of the Web Application DE_strore. Below, it will be also discussed the best suitable technologies chosen in according to this software architecture.

2.4 Django MVC Framework considerations

Django, a high-level Python web framework has been chosen as framework to adopt for the implementation of DE_Store.

Django is by definition an MVC framework, being so designed and structure in order to quickly develop web applications built on top of a model view controller architecture.

As this framework offers natively support from API_Restuful design and testing (by Django rest Framework), and an environment to build and run unit test, incantating so TDD (test driven development), which will be discussed later, Django comes natively with a limitation individual doing the analysis of the framework. In this section, we will analyse native django limitation related to the debase management and solutions provided.

For DE_Store to being able to interact with data, we will need to us a **database connector** (o simply, database connection). A database connector is a software facility that allows a client to interact with a database server software, whether on the same machine or not. Indeed, a connection is required to interact with the database and receive a response.

In Django, the web framework used for the creation of the prototype, multiple database connectors are supported, having al the following structure “django.db.backend.”, followed by the database engineer preferred (SQLite, MySQL, PostgreSQL).

By default, Django uses SQLite as default database engine, indeed, is possible to find on as default value “django.db.backend.sqlite3”, in the default Django configuration settings. Thanks to further analysis taken during the architecture discussion and technologies adoption analysis, because of future optimization and needs, it has been decided to adopt MySQL instead of SQLite for the reasons listed below.

The choice to use MySQL instead of SQLite has been taken in order to allow different application to exploit the same dataset contained within our database server.

Below, can see the **key differences between MariaDB and SQLite**:

MySQL	SQLite
Development in C and C++	Entirely developed in C
Requires a database server to interact with the client over the network, and because of this, is possible to keep the server in a dedicated database server for security reasons.	Is a serverless embedded database that runs as part of the application and cannot connect with any other application over the network.
Support almost all the data type like TYNINT, SMALLINT etc, having a wider management of data resources	Support only BLOB, NULL, INTEGER, TEXT, REAL.

Portability is usually a tedious job and time-consuming process, but thanks to Django, is possible to save and re-run the data migrations to rebuild the database schema.	SQL directly writes data in a file and can move pretty easily. From another side, is not the ideal for an increasing quantity of data.
Can handle multiple simultaneous connections, allowing so to interact with multiple clients (managers) at the same time.	Can cater only one connection at a time.

3.Product Design

3.1 Requirements

The minimal valuable product of the DE_store is expected to offer the following functionalities:

- **Login:** The first version of the Prototype allows an authorized user (managers) to access to the DE_store management platform.
- **User management:**
  - Admin are able to login and create new authorized users by using the built-in Django admin dashboard.
- **Sales preview:**
  - A manager will be able to view a report related to the sales, in particular
    - Total number of sales made
    - Total revenue
    - Average cost for transaction
    - Sales amount today
    - Sales amount in the current month
    - Sales amount from the current year
- **Product management:**
  - View list of products
  - Receive an alert (email), once the product stock items would go under the minimum items.
  - View Information related to the Products, like ID, name, price, description, category, stock, minimum stock, status.
- **Financial Verification:**
  - Emulate the integration of a third-part service that will
- **Offer:**
  - View list of Offers (with name, description and creation date)
  - Create an offer
- **Loyalty Card:**

    - View list of loyalty card with their relative status and information (card number, Owner full name, owner email, points accumulated in the card, activation date, state: active or not active).
- **Logout:**
    - To increase security levels, admins are able to logout once access to the platform is no longer needed and login back once needed.

3.2 Requirement's assumptions

- One store can have **one or more** managers.
- By default (during the item creation process), all the delivery charges are free, but of course, this value is modifiable by the manager
- Just the manager of the store can modify the prices of the shop products.
- The sales performances are associated to the shop.
- Each manager can view just the product of its stop, editing price and minimum stock of the product in according to the store needs.
- One client can have **just one** loyalty card associated to him/her for time.
- One product can have **at most one** offer associated to it for time.

3.3 User Stories

- As a manager, I want to be able to login to administration portal.
- As a manager, I want to be able to logout once the access to the administration portal is no longer needed.
- As a manager, I want to view the list of the product sold in my own warehouse, visualizing the stock status of each product.
- As a manager, I am able to edit the price of each product.
- As a manager, I am able to filter the products in according to the offer currently associated to them.
- As a manager, I am automatically receiving email alerts related to each product where quantity is smaller or equal the minimum stock needed.
- As a manager, I want to view the list of loyalty cards
- As a manager, I want to view the prototype of integration of the financial checker portal
- As a Manager, I want to be able to view the list of offers and create new one for my warehouse.

3.1 Database ER Model

Before to implement the actual models, as part of the design process, an Entity-Relationship model has been built, in order to keep each entity as much independent as possible, avoid duplicated data, Wich leads to recusancy, and increasing number of relationships where needed, in order to have a fully normalized, stable and scalable database.

The ER model produced takes into considerations both requirements and project assumption, in according to the User Experience of the user within the application.

Indeed, as part of the project assumption, a user, is able to insert a list of products inside a char. Because of this, the database, models, and functions associated, takes under considerations that the sales report must be created in according to the payments sum, which are associated to a char with status: ordered.

As in the future we may have multiple types of managers, involved more in the accountant aspect of the warehouse, or more involved in the marketing aspect of it, needing to have access for example just to the product and offer section, each user has an attribute level, which offers scalability, as we could associate to each level a set of permission, having so a granular access to each page of the prototype.

Considering that most of the geo localization tasks nowadays rely on third part services, and for being able in the future to visually display the distributions of orders in according to the user glocalization, the address model (or table), has been designed to store also information like latitude and longitude.

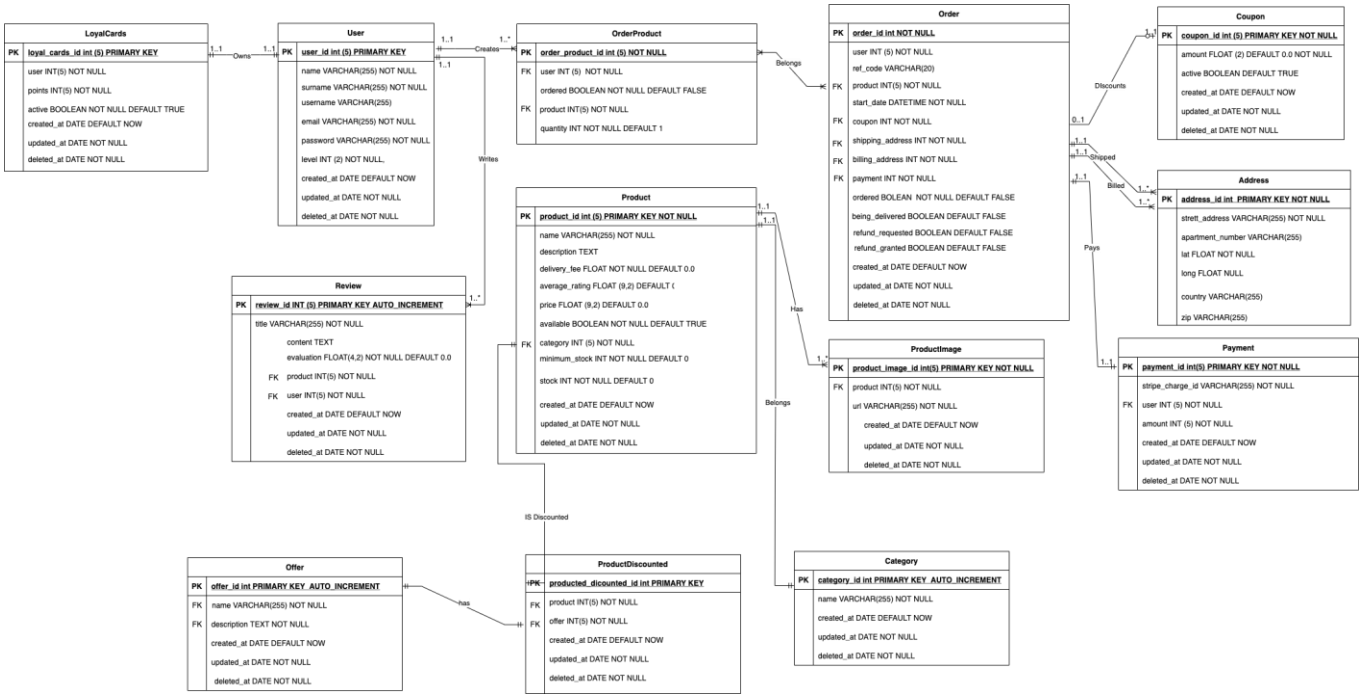


Fig. 11 ER-Model

4.Product Implementation

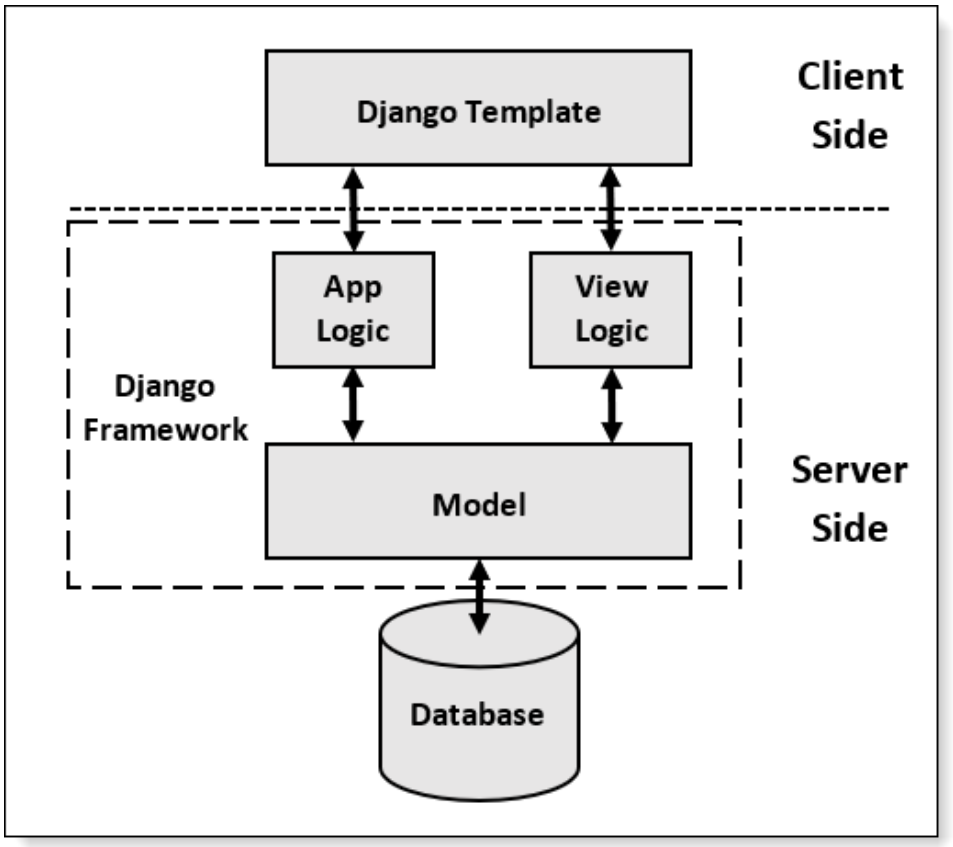
4.1 Technology stack, tools and services

During the development of the DE_store, in according to the software requirements and non-direct requirements, the following technologies and tools has been used to implement our system:

- HTML5, CSS3, JavaScript

- Bootstrap
- Django
- MariaDB Database
- GitHub

The backend web framework used for this project, **Django**, allowed us to implement a **model-view-controller architect.**



Django MVT approach, source: GeeksForGeeks.com

Thanks to the use of Django, it is possible to simply manage error message that could both be shown to the user (for example in the case of wrong credentials), and being registered in the logs.

Using the messages built-in directives, we are also able to differentiate logs in five different levels:

- Information (info)
- Error
- Warning
- Debug
- Success

4.2 Adoption of Celery for stock monitoring

To provide a high standard of supervision, it has been used Celery for the queuing of tasks executed in according to an event.

In our case, a stock status checker task will be run every 60 minutes.

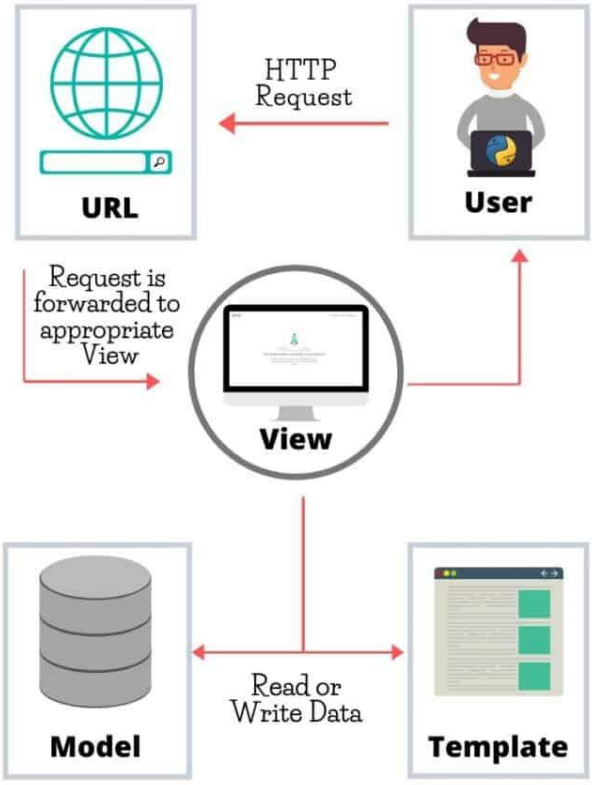
This stock status will:

- Monitor if the current product **stock is under its minimum.**
- **Send an email** to a manager if a product stock is under its minimum.

4.3 Protocols used

For serving to the client each view upon to a request, by default Django uses the HTTP protocol, which would run in its default port (80 / 8080). In case of development mode instead, the default pot used by Django is the 8000, using always HTTP as default protocol.

HTTP is an application layer protocol used to transfer hypermedia files (like HTML document), mostly used for the communication between web browser (clients) and web servers.



Once the product will be released (in production), the **https protocol** will be enforced in any page of the web platform. HTTPS, or also hypertext Transfer protocol Secure, is a secure version of http with uses TLS/SSL to encrypt data. Thanks to https indeed, we will be able to encrypt all data, included in the authentication process needed to access to the website, guarantying so privacy and integrity.

HTTPS runs on a different port (443) compared to HTTP (8080). Indeed, any web server will be able to accept requests on both ports at the same time, and anyone could simply edit the URL address to explicitly use http instead of https. To avoid these issues, is possible to convert each request from http to https by splitty web server set-up or via the use of the third-pard proxy services like Cloudflare.

3. System evaluation

In the following sections, it will be done presented the current system evaluation and possible future improvements.

5.1 Unit test

As part of the system validation, a part from manual testing, in order to avoid regression and unexpected behaviours (or bug), is key to adopt where possible a Test-driven development, a programming development style where each unit test describes a requirement and each failing test would result on a lacks of feature.

As TDD (or Test Driven Development) may initially sounds being time-consuming, the real focus is shifted to the design and implementation so the tests, and each developer would write just enough code to make the test pass, avoiding to have over-engineering methods or not needed elaborations, as unit tests may both not just validate a requirements itself in terms of assertion (verification) of actual result and expected results from an elaboration, but may also tests the size of any artefact created, or time elapsed since the start of a specific tasks, essential for time-constrained requirements.

In Django, all the unit tests may be kept stored in a single file (tests.py), by default created during the project set-up.
Before to commit any new change, it would be necessary to run the command **python3 manage.py test to** run each test.

By using Django, each testcase is initially created as a class, with independent data-setup (or mocking) for testing purposes.
This would make possible to create 1 Class for each Test case, and use one test case for each functionality of the product.

```python
from django.test import TestCase

# Create your tests here.
from .models import *
class LoyalCardTestCase(TestCase):
    def setUp(self):
        User.objects.create(name="Davide", username="bob99", email="d@gmail.com", password="cc", level=1, created_at='2021-03-03')
        new_user =  User.objects.get(email="d@gmail.com")
        LoyalCard.objects.create(user=new_user, points=200, active=True)

    def test_card_being_disabled(self):
        user =  User.objects.get(email="d@gmail.com")
        card = LoyalCard.objects.get(user=user)
        LoyalCard.disable_card(card.id)
        card = LoyalCard.objects.get(user=user)
        self.assertEqual(card.active, False)


    def test_card_being_enabled(self):
        user =  User.objects.get(email="d@gmail.com")
        card = LoyalCard.objects.get(user=user)
        LoyalCard.enable_card(card.id)
        card = LoyalCard.objects.get(user=user)
        self.assertEqual(card.active, True)
```

5.2 Future Improvements

The infrastructure is designed and implemented, and as we have seen, is fairly easy to run all the unit tests of the application.

In order to delivery any new version of the application beign sure that the new changes would not introduce bugs, is useful to implement a pipeline that would run a set of checks and tests every time a new pull requests are created, in order to authorize the merge of a new feature and fixes, just if all the tests pass.

This would prevent us to introduce further bugs while we would update a requirement, add new ones or change portions of code during refactory operations.

6. Conclusion

It has been shown which software architecture would suit best to the implementation fo the DE_store.

It has been also shown how fundamental is to choose a technology or framework (as Django in our case), and also evaluate if any default configuration of the framework chosen can be used changed in order to improve the availability, scalability and security of the product. For this case, we have seen how important would be for DE_store to switch from SQLite3 to MySQL or MariaDB.

Once have decided which changes to adopt in order to get best performances from the framework, it has been shown how beneficial could be to establish a software development methodology, as TDD (Test Driven Development).

We can now conclude summering that:

- MVC is the best architecture in according to our needs.
- Django Rest Framework would allow to exposes API in order to connect the UI and Controller of the application.
- SQLite is not an ideal choice as it would not possible otherwise to keep the application and database in two physical or logical server, and have multiple database connections active at the same time.
- Test Driven Development will slightly delay the start of implementation of actual source code, by instead guarantying that each requirement will be fully satisfised and compliance in according to original expectations, allowing us to quickly view errors if introduced by code changes in the future.

Appendix

Fig. 1 Homepage

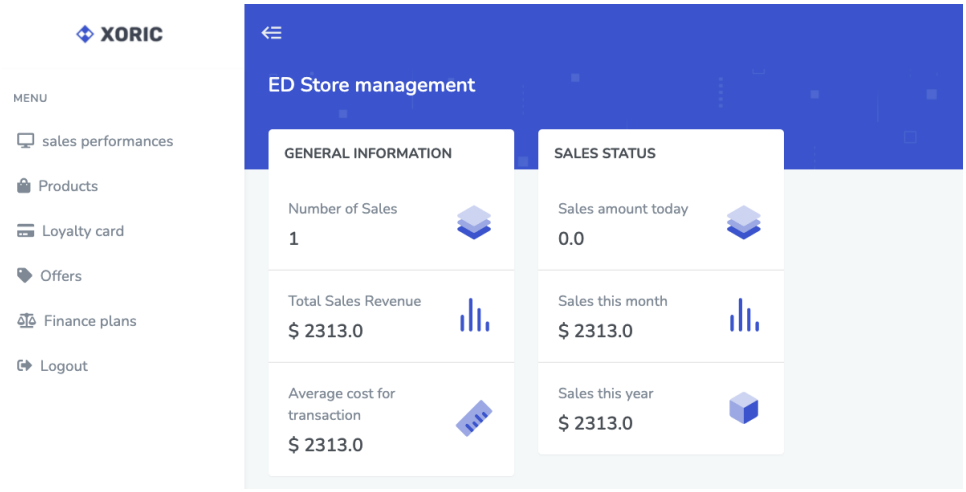


Fig.2 Products section

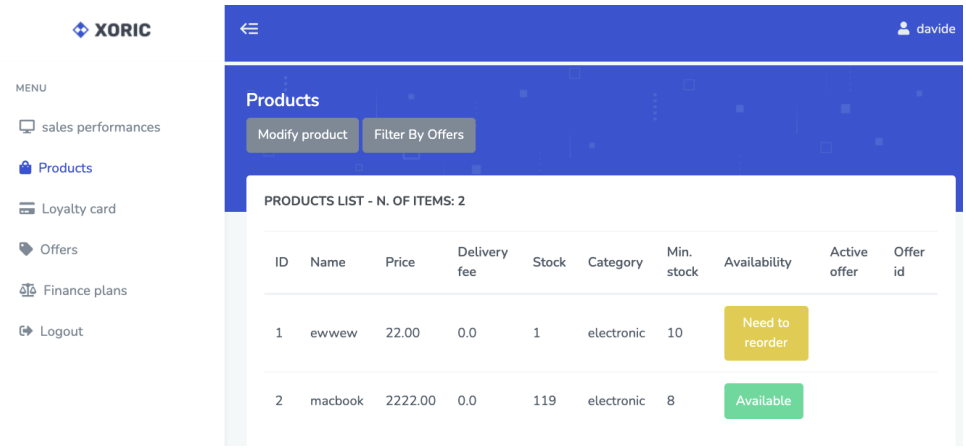


Fig. 3 Edit Product Price

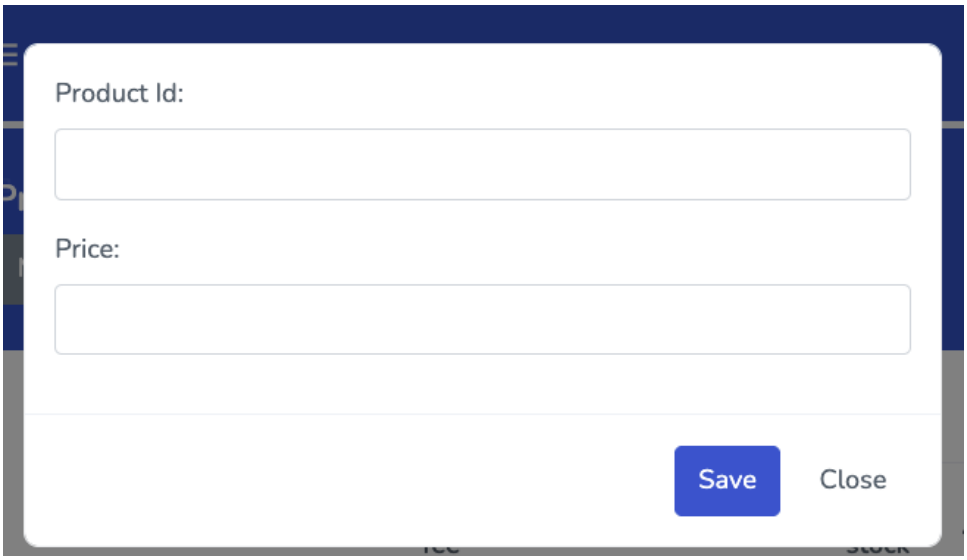


Fig.4 Loyalty card section

Fig. 5 Offers management section

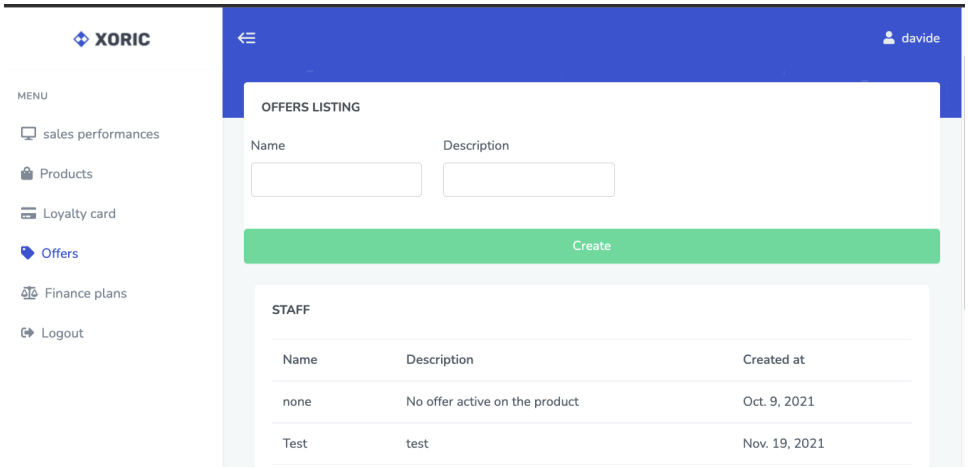


Fig. 6 Financial check section

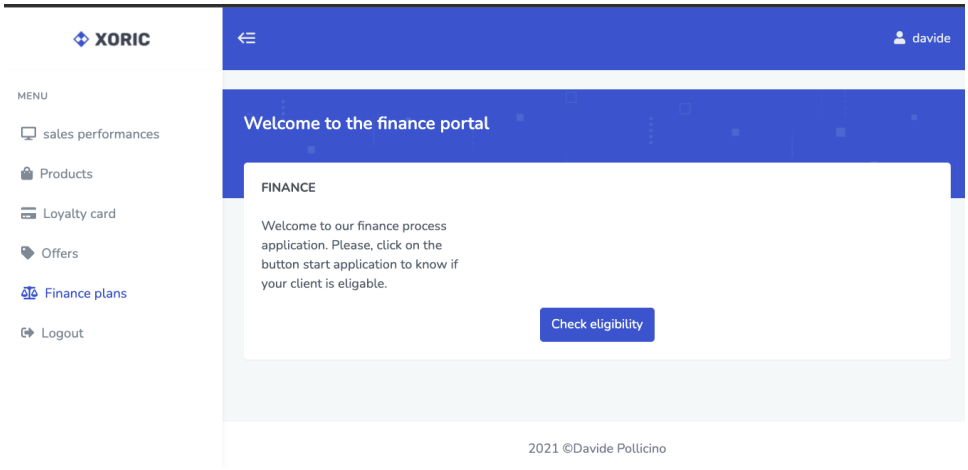


Fig. 7 Login page

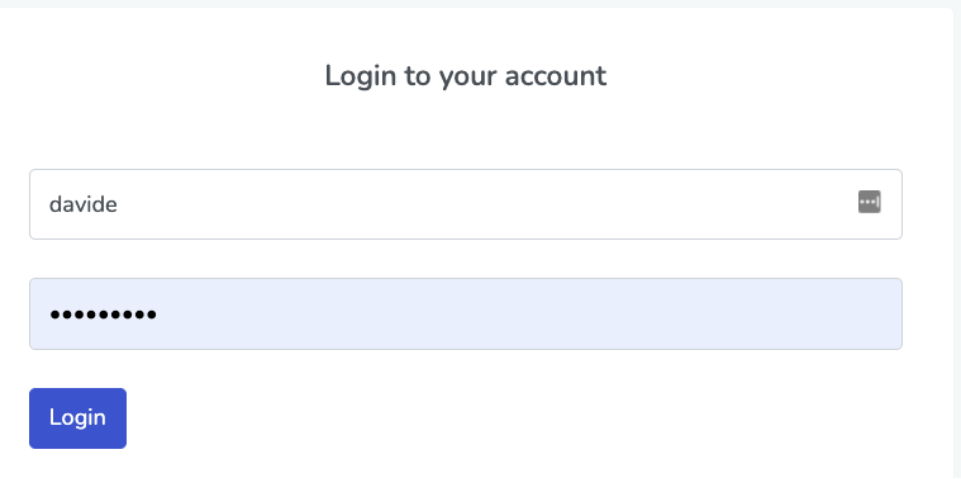


Fig. 8 GitHub Branches management

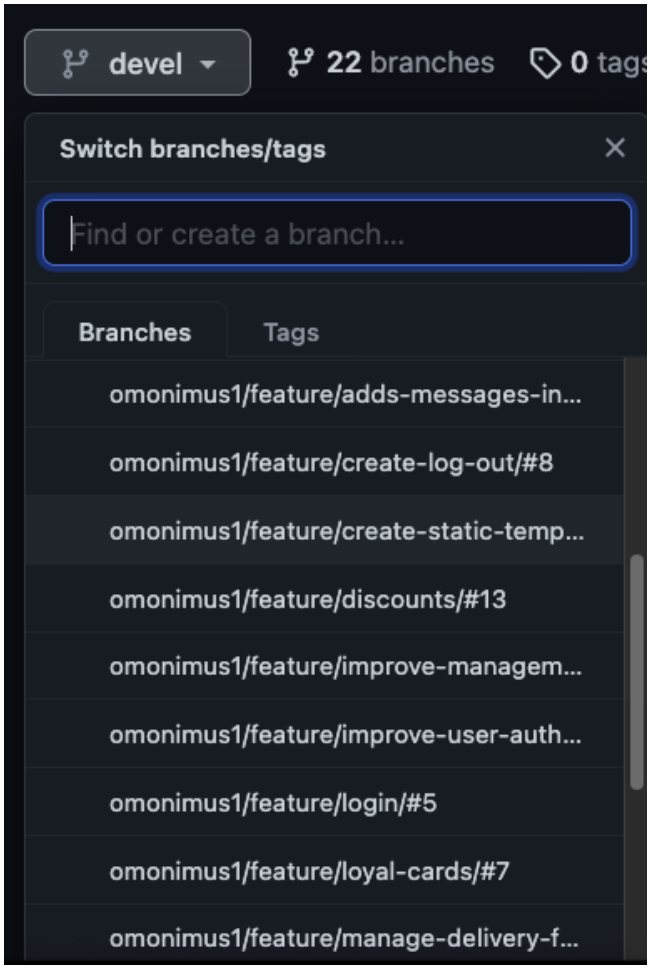


Fig. 9 Use of Pull requests for Code management on GitHub

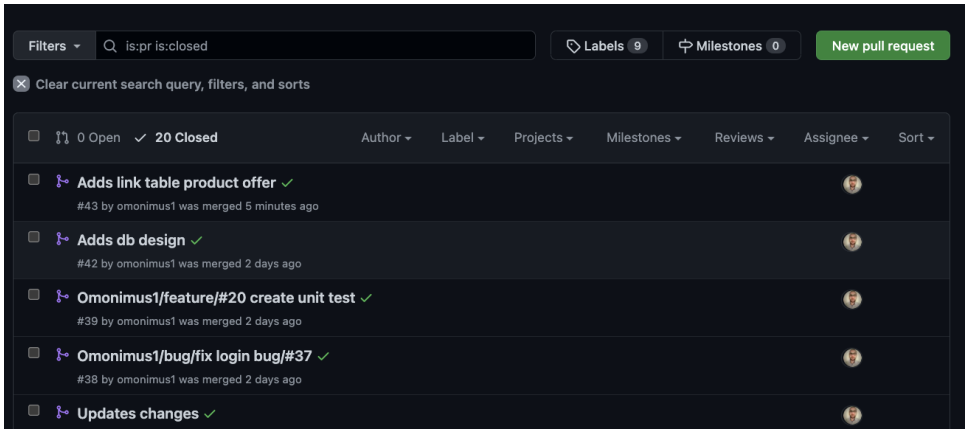


Fig.10 GitHub Issues

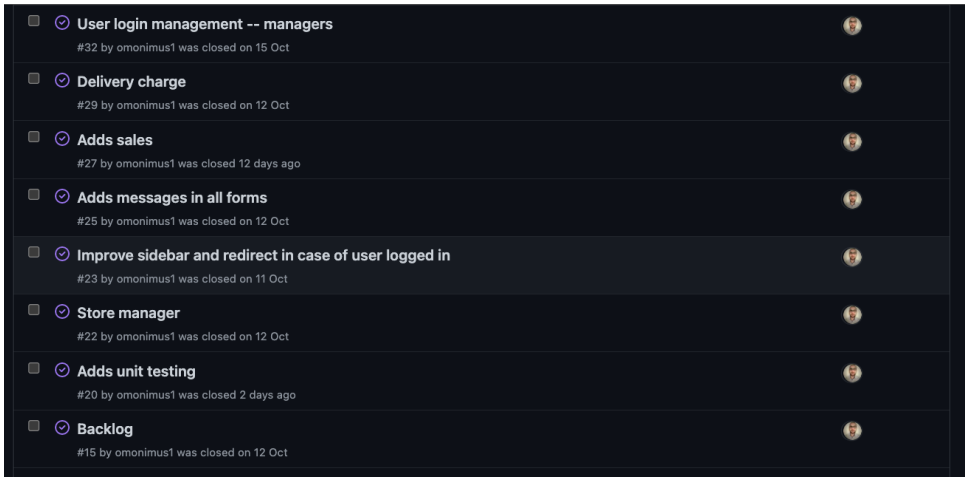


Fig. 11 ER-model

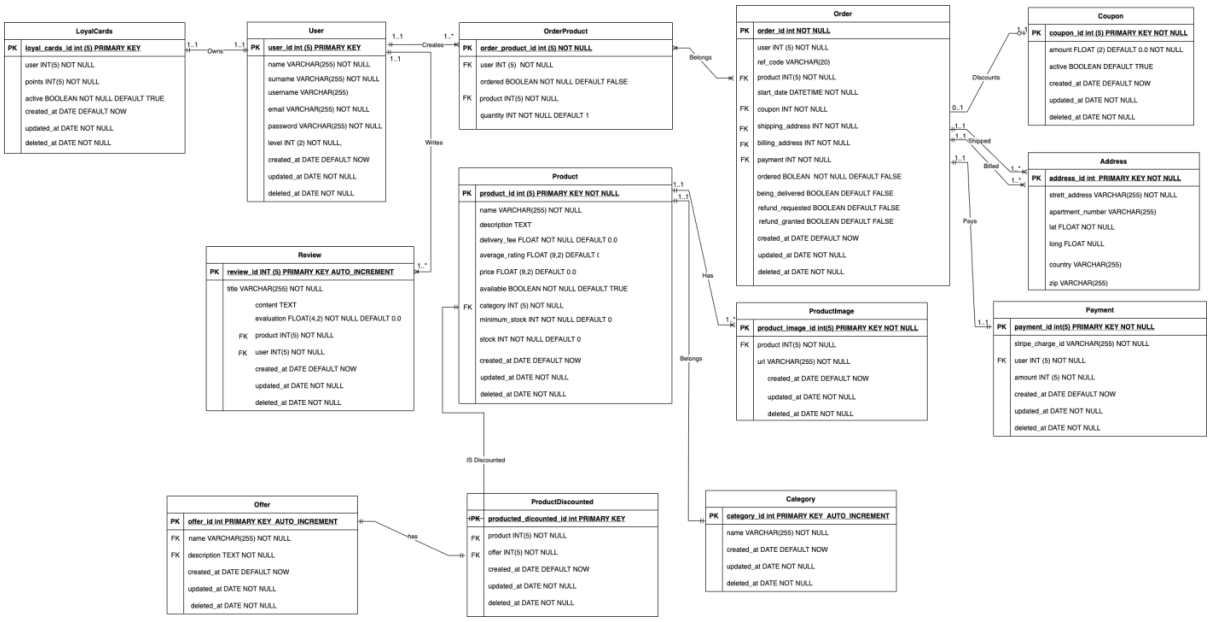


Fig. 12 Unit test

```python
from django.test import TestCase

# Create your tests here.
from .models import *
class LoyalCardTestCase(TestCase):
    def setUp(self):
        User.objects.create(name="Davide", username="bob99", email="d@gmail.com", password="cc", level=1, created_at='2021-03-03')
        new_user =  User.objects.get(email="d@gmail.com")
        LoyalCard.objects.create(user=new_user, points=200, active=True)

    def test_card_being_disabled(self):
        user =  User.objects.get(email="d@gmail.com")
        card = LoyalCard.objects.get(user=user)
        LoyalCard.disable_card(card.id)
        card = LoyalCard.objects.get(user=user)
        self.assertEqual(card.active, False)


    def test_card_being_enabled(self):
        user =  User.objects.get(email="d@gmail.com")
        card = LoyalCard.objects.get(user=user)
        LoyalCard.enable_card(card.id)
        card = LoyalCard.objects.get(user=user)
        self.assertEqual(card.active, True)
```

Education Use Content

I hereby grant my permission for this project to be stored, distributed and shown to other Edinburgh Napier university students and staff for educational purpose.

Signature: Davide Pollicino