

**Module Code & Module Title**

**CS6P05NT-Final Year Project**

**Assessment Weightage & Type**

**Final Year Project Report**

**Year and Semester**

**2024-25 Autumn**

**Student Name:**

**London Met ID:**

**College ID:**

**Assignment Submission Date: 2025/04/30**

**Internal Supervisor:**

**External Supervisor:**

**Title:**

*I confirm that I understand my coursework needs to be submitted online via Google Classroom under the relevant module page before the deadline for my assignment to be accepted and marked. I am fully aware that late submissions will be treated as non-submission and a mark of zero will be awarded*

**Acknowledgement**

First, I would like to express my gratitude to **Itahari International College** and all the faculty members for providing such a fantastic opportunity for my final year project. This project has been an incredible educational experience for my future, and I am extremely thankful to everyone involved.

I would like to begin by extending my thanks to **AFC Food Restaurant**, who has been a supportive client and provided me with all the necessary requirements to develop the system effectively. Their assistance has been instrumental in building the project correctly.

Additionally, I want to acknowledge and appreciate my supervisors**, Mr. Mero Raja Prasad Pradhan, and Mr. Ujjwal Subedi**. Without their guidance and support throughout the project, it would never have been completed. Their invaluable assistance and mentorship have been truly invaluable throughout this journey.

Lastly, I would like to emphasize that working on this project has been an outstanding educational opportunity for me. I am pleased with the remarkable results that have been achieved, and I attribute this success to the collective efforts of Itahari International College, my professors, instructors, and everyone involved. I sincerely appreciate the hard work and dedication of everyone involved in making this project a progressive success.

**Abstract**

The final year project "VSITA()” aims to address the issues faced by small restaurants and cafes in Nepal, including slowdowns and misunderstandings in the food ordering process, high levels of manpower required, and long wait times for orders and billing. The project proposes the development of a digital ordering system, called "Foodiefy", which allows customers to place orders directly from their devices and enables restaurant staff to see and fulfil the orders.

Foodiefy also provides instant billing, allowing customers to view and pay for their total bill on their phones, divide costs among their group, and leave reviews of individual dishes. The project aims to improve the convenience and customer service of the restaurant, as well as provide valuable business insights through analysis of food reviews and sales data. Overall, this system aims to enhance the efficiency and profitability of small restaurants and cafes in Nepal through the use of digital technologies.

**Table of Contents**

[1 Introduction 1](#_Toc196149129)

[1.1 Project Description 1](#_Toc196149130)

[1.2 Current Scenario 1](#_Toc196149131)

[1.3 Problem Domain and Project as a Solution 1](#_Toc196149132)

[1.4 Aim and Objectives 1](#_Toc196149133)

[1.5 Structure of The Report 1](#_Toc196149134)

[1.5.1 Background 1](#_Toc196149135)

[1.5.2 Development 1](#_Toc196149136)

[1.5.3 TESTING AND ANALYSIS 1](#_Toc196149137)

[1.5.4 CONCLUSION 1](#_Toc196149138)

[2 BACKGROUND 2](#_Toc196149139)

[2.1 ABOUT THE END USERS 2](#_Toc196149140)

[2.2 UNDERSTANDING THE SOLUTION (Technical Terms related to the project) 2](#_Toc196149141)

[2.3 SIMILAR PROJECTS 2](#_Toc196149142)

[2.4 COMPARISONS (Comparing the features and critical evaluation of the solution) 2](#_Toc196149143)

[3 Development 3](#_Toc196149144)

[3.1 CONSIDERED METHODOLOGIES 3](#_Toc196149145)

[3.2 SELECTED METHODOLOGY 3](#_Toc196149146)

[3.3 PHASES OF METHODOLGY 3](#_Toc196149147)

[3.4 SURVEY RESULTS 3](#_Toc196149148)

[3.4.1 PRE-SURVEY RESULTS 3](#_Toc196149149)

[3.4.2 POST-SURVEY RESULTS 3](#_Toc196149150)

[3.5 Requirement Analysis 3](#_Toc196149151)

[3.6 DESIGN 3](#_Toc196149152)

[3.7 Implementation 3](#_Toc196149153)

[3.7.1 Database Design 3](#_Toc196149154)

[3.7.2 Database Transition: From SQLite to MySQL 8](#_Toc196149155)

[3.7.3 Web Application 26](#_Toc196149156)

[4 Testing and Analysis 26](#_Toc196149157)

[4.1 TEST PLAN 26](#_Toc196149158)

[4.1.1 UNIT TESTING, TEST PLAN 26](#_Toc196149159)

[4.1.2 SYSTEM TESTING, TEST PLAN 26](#_Toc196149160)

[4.2 Unit Testing 26](#_Toc196149161)

[4.3 System Testing 26](#_Toc196149162)

[4.4 CRITICAL ANALYSIS 27](#_Toc196149163)

[5 CONCLUSION 28](#_Toc196149164)

[5.1 LEGAL, SOCIAL AND ETHICAL ISSUES 28](#_Toc196149165)

[5.1.1 LEGAL ISSUES 28](#_Toc196149166)

[5.1.2 SOCIAL ISSUES 28](#_Toc196149167)

[5.1.3 ETHICAL ISSUES 28](#_Toc196149168)

[5.2 ADVANTAGES 28](#_Toc196149169)

[5.3 LIMITATIONS 28](#_Toc196149170)

[5.4 FUTURE WORK 28](#_Toc196149171)

[6 References 29](#_Toc196149172)

[7 BIBLIOGRAPHY 30](#_Toc196149173)

[8 APPENDIX 31](#_Toc196149174)

[8.1 APPENDIX A: PRE-SURVEY 31](#_Toc196149175)

[8.1.1 PRE-SURVEY FORM 31](#_Toc196149176)

[8.1.2 SAMPLE OF FILLED PRE-SURVEY FORMS 31](#_Toc196149177)

[8.1.3 PRE-SURVEY RESULT 31](#_Toc196149178)

[8.2 APPENDIX B: POST-SURVEY 31](#_Toc196149179)

[8.2.1 POST-SURVEY FORM 31](#_Toc196149180)

[8.2.2 SAMPLE OF FILLED POST-SURVEY FORMS 31](#_Toc196149181)

[8.2.3 POST-SURVEY RESULT 31](#_Toc196149182)

[8.3 APPENDIX C: SAMPLE CODES 31](#_Toc196149183)

[8.3.1 SAMPLE CODE OF THE UI 31](#_Toc196149184)

[8.3.2 SAMPLE CODE FOR THE AUTOMATION SCRIPT 31](#_Toc196149185)

[8.4 APPENDIX D: DESIGNS 31](#_Toc196149186)

[8.4.1 GANTT CHART 31](#_Toc196149187)

[8.4.2 WORK BREAKDOWN STRUCTURE 31](#_Toc196149188)

[8.4.3 ALGORITHMS & FLOWCHARTS 32](#_Toc196149189)

[8.4.4 3D MODELLING 32](#_Toc196149190)

[8.4.5 HARDWARE ARCHITECTURE 32](#_Toc196149191)

[8.4.6 DATA FLOW DIAGRAMS 32](#_Toc196149192)

[8.4.7 USE CASE 32](#_Toc196149193)

[8.4.8 WIREFRAME 32](#_Toc196149194)

[8.5 APPENDIX E: SCREENSHOTS OF THE SYSTEM 32](#_Toc196149195)

[8.6 APPENDIX F: USER FEEDBACK 32](#_Toc196149196)

[8.6.1 USER FEEDBACK FORM 32](#_Toc196149197)

[8.6.2 SAMPLE OF FILLED USER FEEDBACK FORMS 32](#_Toc196149198)

[8.7 APPENDIX G: FUTURE WORK 32](#_Toc196149199)

[8.7.1 READINGS FOR FUTURE WORK 32](#_Toc196149200)

**Table of Figures**

[Figure 1 Custom User Model 4](#_Toc196146770)

[Figure 2 Customer User Model Definition in settings.py 4](#_Toc196146771)

[Figure 3 UploadedPCAPS Model 5](#_Toc196146772)

[Figure 4 Rule Model 6](#_Toc196146773)

[Figure 5 MatchedPacket Model 7](#_Toc196146774)

[Figure 6 Logs Model 8](#_Toc196146775)

[Figure 7 Docker Desktop Containers Page 9](#_Toc196146776)

[Figure 8 Docker Desktop Images Section 10](#_Toc196146777)

[Figure 9 Running the Docker Images Command 10](#_Toc196146778)

[Figure 10 Docker Hub Search Result 11](#_Toc196146779)

[Figure 11 Docker Pull Result 11](#_Toc196146780)

[Figure 12 MySQL Image Listing Success 12](#_Toc196146781)

[Figure 13 Docker Desktop Image Listing Success 12](#_Toc196146782)

[Figure 14 Docker Run Result 13](#_Toc196146783)

[Figure 15 Container Displayed in Docker Desktop 13](#_Toc196146784)

[Figure 16 Docker PS Command Result 13](#_Toc196146785)

[Figure 17 MySQL Workbench Home Page 14](#_Toc196146786)

[Figure 18 Inputting Database Details 14](#_Toc196146787)

[Figure 19 Database Testing Connection Result 15](#_Toc196146788)

[Figure 20 New Connection Profile 15](#_Toc196146789)

[Figure 21 MySQL Workbench Database Home Page 16](#_Toc196146790)

[Figure 22 Listing all Database Schemas 16](#_Toc196146791)

[Figure 23 Creating New Schema 17](#_Toc196146792)

[Figure 24 Schema Creation Query 17](#_Toc196146793)

[Figure 25 Schema Creation Confirmation 17](#_Toc196146794)

[Figure 26 Empty Database Creation Success 18](#_Toc196146795)

[Figure 27 Django Directory Structure Before Data dump 18](#_Toc196146796)

[Figure 28 Directory Structure After Running Data dump Command 19](#_Toc196146797)

[Figure 29 Contents of db.json Dump File 20](#_Toc196146798)

[Figure 30 Default Database Configuration 20](#_Toc196146799)

[Figure 31 Updated Databases Configurations 21](#_Toc196146800)

[Figure 32 Running Django Application 21](#_Toc196146801)

[Figure 33 Migration Result 22](#_Toc196146802)

[Figure 34 Created Tables in Database 23](#_Toc196146803)

[Figure 35 Result of Loading Data 24](#_Toc196146804)

[Figure 36 User Data in SQLite Database 24](#_Toc196146805)

[Figure 37 User Data in MySQL Database 24](#_Toc196146806)

[Figure 38 System Testing 26](#_Toc196146807)

**Table of Tables**

No table of figures entries found.

**Line breaks here**

# Introduction

## Project Description

## Current Scenario

## Problem Domain and Project as a Solution

## Aim and Objectives

## Structure of The Report

### Background

### Development

### TESTING AND ANALYSIS

### CONCLUSION

# BACKGROUND

## ABOUT THE END USERS

## UNDERSTANDING THE SOLUTION (Technical Terms related to the project)

## SIMILAR PROJECTS

## COMPARISONS (Comparing the features and critical evaluation of the solution)

# Development

## CONSIDERED METHODOLOGIES

## SELECTED METHODOLOGY

## PHASES OF METHODOLGY

## SURVEY RESULTS

### PRE-SURVEY RESULTS

### POST-SURVEY RESULTS

## Requirement Analysis

## DESIGN

## Implementation

Following the selected SDLC methodology, the implementation of this Final Year Project has been done according to the timeline presented in the Gantt chart which is available in the appendix section of this document.

### Database Design

After making the initial ERD which is present in the design phases section of this document and the continuous iterations and refinements to the models throughout this project build lifecycle, the sufficient model class for the database for the application was developed.

The model classes of different Django applications used in this project is attached below.

#### Model Description and Images

**Custom User Model:**

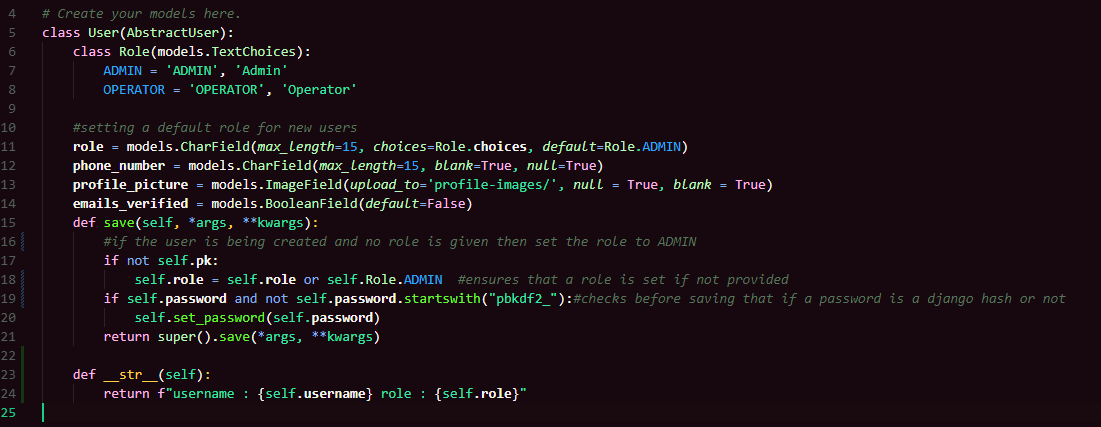


Figure Custom User Model

To meet the requirements of this project, a custom user model was implemented by extending Django’s built-in AbstractUser class. This allowed the addition of extra fields and functionality tailored to the application. The model includes a role field with predefined choices (Admin and Operator) to distinguish user access levels, with Admin set as the default for newly created users. Additional fields such as phone\_number, profile\_picture, and emails\_verified were introduced to support extended user management features.

The default save() method was overridden in this User model class to ensure secure password handling and to enforce role assignment during user creation, mitigating the risk of unforeseen bugs or issues during testing or in production environments. After defining the custom user model in the application’s models.py file, it was registered in the project's settings.py file by setting the AUTH\_USER\_MODEL to point to the custom user model, ensuring compatibility and avoiding conflicts with Django’s default authentication system.

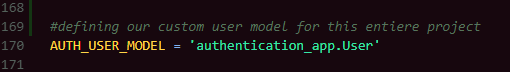


Figure Customer User Model Definition in settings.py

**UploadedPCAPS Model:**

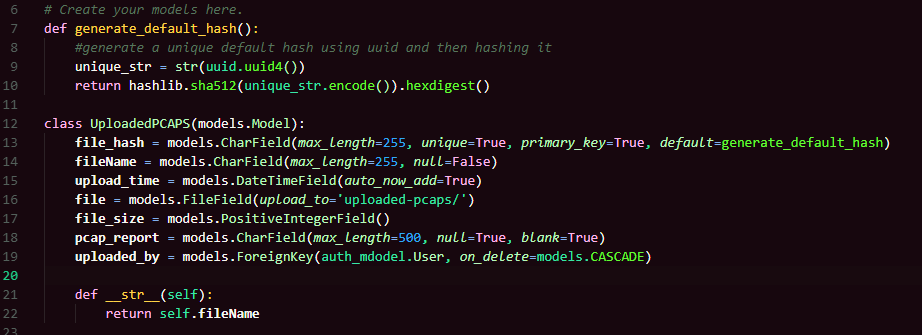
****

Figure UploadedPCAPS Model

The UploadedPCAPS model is responsible for handling and storing metadata related to PCAP (Packet Capture) files uploaded by users. Each entry includes a file\_hash, which serves as the primary key and ensures the uniqueness of uploaded files. Ideally, this hash is calculated at the view level using the **calculate\_file\_hash()** function, which reads the file in chunks and computes a SHA-512 hash for reliable identification and duplicate detection.

However, to ensure robustness and maintain workflow continuity even in the case of unexpected errors or bugs in the view logic, the model includes a fallback mechanism using the **generate\_default\_hash()** function. This function generates a unique hash based on a UUID, which is then hashed using SHA-512. This guarantees that a valid and unique file\_hash is always available when a record is created.

Other fields in the model include fileName, upload\_time, the actual uploaded file (file), its size in bytes (file\_size), and an optional pcap\_report field to store analysis outcomes. The uploaded\_by field is a foreign key linking to the custom User model, allowing user-based tracking of uploads. The model also defines a \_\_str\_\_() method that returns the file name, improving readability in Django’s admin and query interfaces.

**Rule Model:**

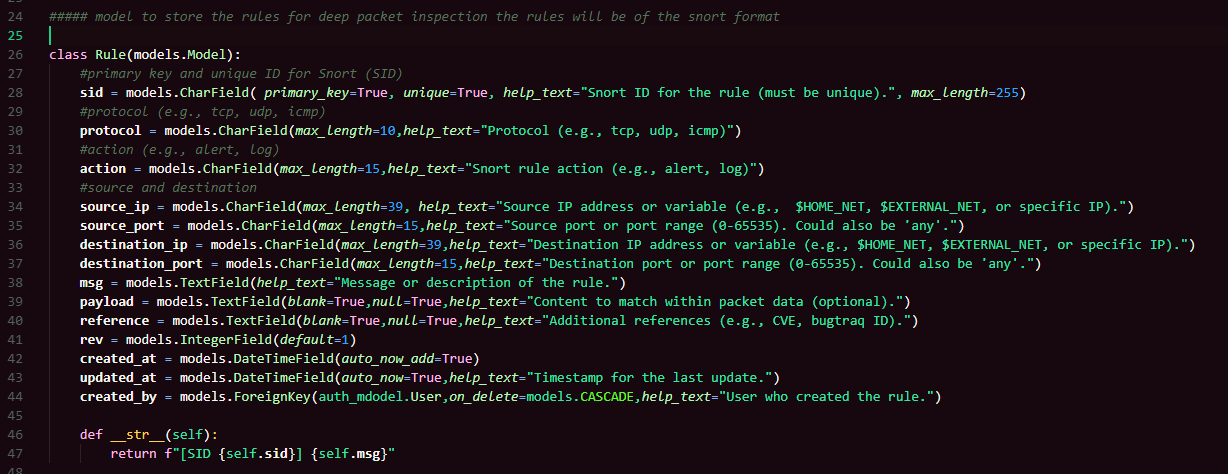


Figure Rule Model

The Rule model is designed to store and manage custom Snort intrusion detection rules within the application. Each rule is uniquely identified by the sid (Snort ID), which is both a primary key and a unique field to prevent duplication. The model captures essential attributes required to define a Snort rule, including the protocol (e.g., TCP, UDP, ICMP), action (such as alert or log), and source/destination IP addresses and ports. These fields allow users to specify traffic patterns the rule should match against.

Additional metadata includes a msg field to describe the purpose of the rule and an optional payload field to define specific content patterns to inspect within packet data. The reference field allows linking external references such as CVEs or bug tracking IDs to provide context or traceability. Each rule also has a revision number (rev) to support version tracking. The created\_by field is a foreign key referencing the custom User model, ensuring that each rule is associated with the user who created it. Timestamp fields (created\_at and updated\_at) automatically record when a rule is added or modified. The model's \_\_str\_\_() method returns a readable format combining the SID and message for clarity in admin views and logs.

**MatchedPacket Model:**

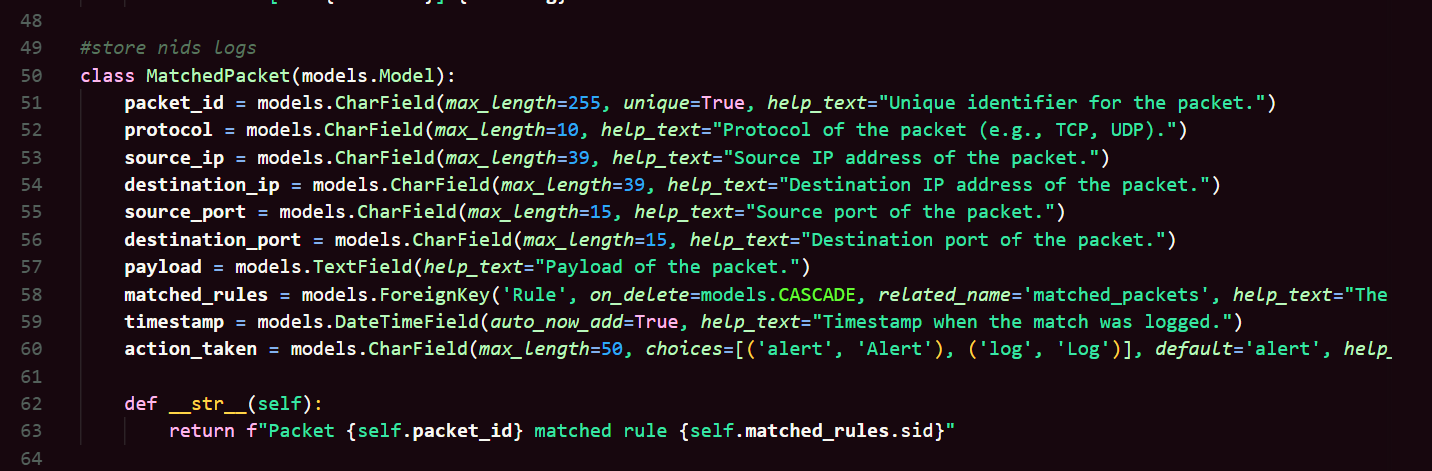
****

Figure MatchedPacket Model

The MatchedPacket model is designed to store information about packets that have triggered a match with a Snort rule. Each packet is uniquely identified by the packet\_id, ensuring that no two packets share the same identifier. The model captures key details of the packet, including its protocol (e.g., TCP, UDP), source and destination IP addresses (source\_ip and destination\_ip), and source and destination ports (source\_port and destination\_port). The payload field holds the packet’s content, which can be used for further analysis or inspection.

A foreign key relationship is established with the Rule model via the matched\_rules field, which indicates the rule that caused the packet match. This allows for easy traceability of the rule that identified the packet. The timestamp field automatically records when the match is logged, and the action\_taken field stores the action performed after the match (either 'alert' or 'log'). The model’s \_\_str\_\_() method returns a readable string combining the packet's unique identifier and the SID of the matching rule, making it easier to reference in logs and administrative views.

**Logs Model:**

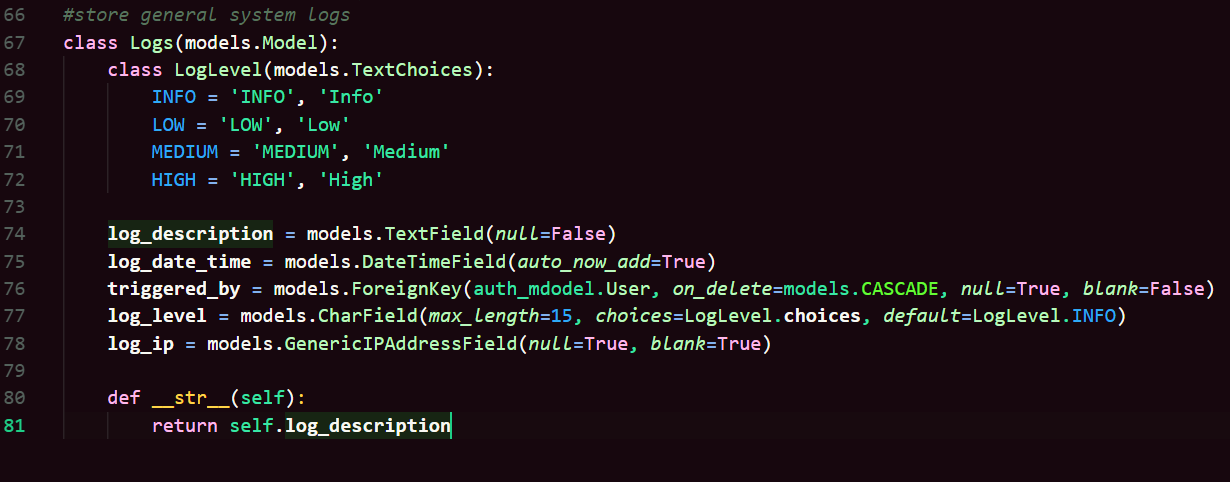


Figure Logs Model

The Logs model is used to store general system logs, capturing critical information related to the operations of the application. It includes a log\_description field that provides detailed information about the logged event, and a log\_date\_time field that automatically records the timestamp when the log is created. This timestamp helps in tracking and analyzing logs over time.

The model utilizes a log\_level field with predefined choices (Info, Low, Medium, High) to categorize the severity or importance of each log entry, allowing for better filtering and prioritization of log data. The triggered\_by field is a foreign key referencing the custom User model, linking the log entry to the user who initiated the event. Additionally, the log\_ip field stores the IP address from which the event was triggered, if available. This is useful for security monitoring and tracking user activity across different devices. The \_\_str\_\_() method returns the log description, providing a clear and concise representation of each log entry, making it easier to view and analyze in administrative interfaces.

### Database Transition: From SQLite to MySQL

Initially, SQLite was used during the development phase for simplicity and ease of use. However, in accordance with the project proposal, the database was later migrated to MySQL for testing and deployment purposes.

For the transition from SQLite to MySQL the following steps were taken:

1. **Installed docker:**

There are multiple ways to run MySQL on Windows, including using tools like XAMPP, WAMP, or directly installing MySQL via an installer. However, solutions like XAMPP, although easy to set up, can be unreliable in the long term—often facing issues like service crashes, port conflicts, and inconsistent behaviour during updates or system reboots.

To ensure a stable and isolated environment, Docker was chosen as the primary method to run MySQL. Docker allows containerized deployment of MySQL, which ensures that the database runs in a consistent and controlled environment regardless of the underlying operating system. This approach eliminates dependency issues, simplifies version management, and improves reliability, making it more suitable for both development and production use cases.

By using Docker, the setup also becomes more portable and can be easily shared or deployed across different systems without configuration mismatches.

A screenshot of a computer

AI-generated content may be incorrect.

Figure Docker Desktop Containers Page

A screenshot of a computer

AI-generated content may be incorrect.

Figure Docker Desktop Images Section

1. **Pulled the MySQL latest docker image from Docker Hub:**

To obtain the latest version of the MySQL image, I visited Docker Hub—an online repository that hosts a wide variety of Docker images for public use. Docker Hub serves as a central registry where developers can search, share, and download container images for different applications and services, including databases like MySQL.

From Docker Hub, I located the official MySQL image and used the docker pull command to download it. This ensures that the environment runs the most up-to-date and secure version of MySQL, maintained directly by the MySQL development team. Using the official Docker image also guarantees better compatibility with Docker containers and reduces potential issues related to version mismatches or misconfigurations.

A black screen with white text

AI-generated content may be incorrect.

Figure Running the Docker Images Command

In the image above I ran the docker images command before visiting the docker hub website to see all the images that is installed in my system, it returned nothing meaning no docker image is installed in my system.

A screenshot of a computer

AI-generated content may be incorrect.

Figure Docker Hub Search Result

In the docker hub we can search for specific images if it is not found in the images section, so I searched for MySQL and it returned the MySQL image page as seen in the image above, we can also see that the page has a copy section, as highlighted in the image above, we need to copy the sentence and then paste it in our terminal.

A screenshot of a computer

AI-generated content may be incorrect.

Figure Docker Pull Result

After the successful of docker pull command that we copied now we need to check if our device can recognize the image and list, it when we run docker images command.

A screenshot of a computer

AI-generated content may be incorrect.

Figure MySQL Image Listing Success

Additionally, the image appeared in the Docker Desktop application, visually confirming the presence of the MySQL image.

A screenshot of a computer

AI-generated content may be incorrect.

Figure Docker Desktop Image Listing Success

1. **Ran the docker container using command line.**

Once the MySQL image was successfully pulled from Docker Hub, the next step was to run a MySQL container instance using the terminal. While tools like Docker Desktop provide a graphical interface for container management, using the command line offers more flexibility and control, especially for configuring ports, environment variables, and volumes.

To start the container, I used the following command:

“*docker run --name fyp-mysql-container -e MYSQL\_ROOT\_PASSWORD=root -p 3306:3306 -d mysql*”

Explanation of the command:

* --name fyp-mysql-container: Assigns a custom name to the container for easier reference.
* -e MYSQL\_ROOT\_PASSWORD=root: Sets the root password for the MySQL server.
* -p 3306:3306: Maps the container's internal port 3306 (MySQL default) to the same port on the host system, allowing local applications (e.g., Django, MySQL Workbench) to connect to it.
* -d: Runs the container in detached mode (in the background).
* mysql: Refers to the image we pulled earlier from Docker Hub.

After executing this command, the container started running in the background with a fully functional MySQL server. This approach provided a reliable, isolated, and easily manageable environment to host the database for my Django project.

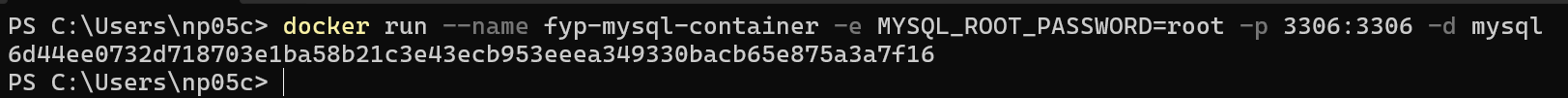


Figure Docker Run Result

After executing the above command, in the image above we can see that it returned a string of characters, it is the container id for the docker image that we ran, it indicates that our image has been successfully built into a container.

A screenshot of a computer

AI-generated content may be incorrect.

Figure Container Displayed in Docker Desktop

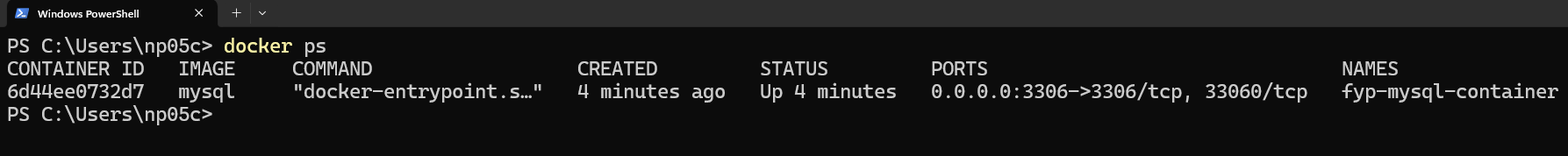


Figure Docker PS Command Result

1. **Used MySQL Workbench to visually inspect and manage the database.**

After the image has been successfully created, into a container now we can connect to it using MySQL Workbench, it is a tool used to connect to a MySQL database and visually manage it.

A screenshot of a computer

AI-generated content may be incorrect.

Figure MySQL Workbench Home Page

To add a new connection, we need to click on the MySQL Connections plus(+) icon and then a new window will appear, we need to input our database details there.

A screenshot of a computer

AI-generated content may be incorrect.

Figure Inputting Database Details

After that we need to check if our passwords matches or if our database is accessible or can be connected to or not, to do that we need to click on the test connection button, it will either be success or failure.

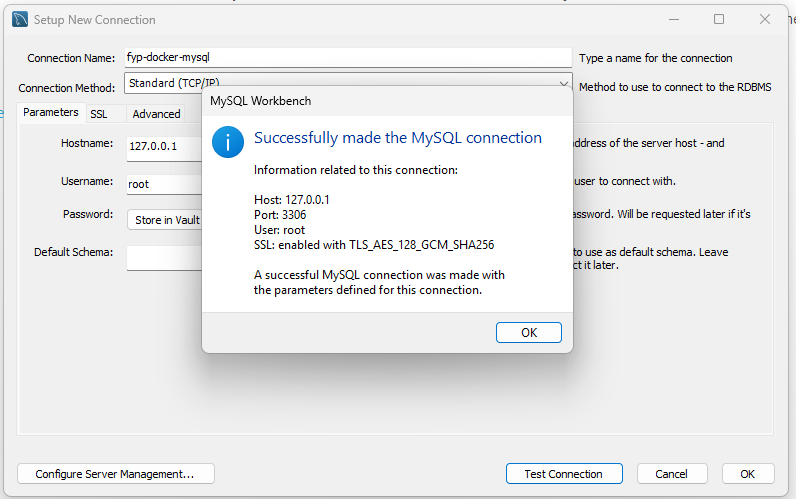


Figure Database Testing Connection Result

We successfully connected to the database table now we need to click ok and then a new connection profile will be created at the MySQL Workbench home page.

A screenshot of a computer

AI-generated content may be incorrect.

Figure New Connection Profile

If we click our connection profile that is being displayed, we can now connect and interact with our database using MySQL Workbench.

A screenshot of a computer

AI-generated content may be incorrect.

Figure MySQL Workbench Database Home Page

Django doesn’t automatically create the database that is specified in the settings.py so we need to create an empty database of the same name at first before applying any migrations.

A screenshot of a computer

AI-generated content may be incorrect.

**B**

**A**

Figure Listing all Database Schemas

From the image above we can see that the label “A” shows the list of all the databases, by default MySQL has only the sys database that stores all the information about users and table schemas and other vital administrator information. To add a new database schema for our project we need to click on label “B” to add the database, by doing that a new window will appear which will ask us to create a name and then press apply. It will show us the command that will be used to create a new schema(database) and then press apply again.

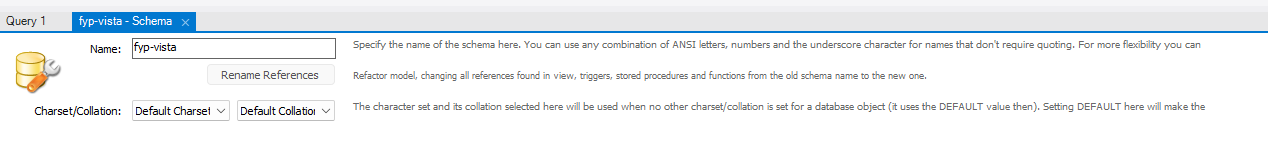


Figure Creating New Schema

A screenshot of a computer

AI-generated content may be incorrect.

Figure Schema Creation Query

After that a new window will appear we need to click finish and then our new database will be created.

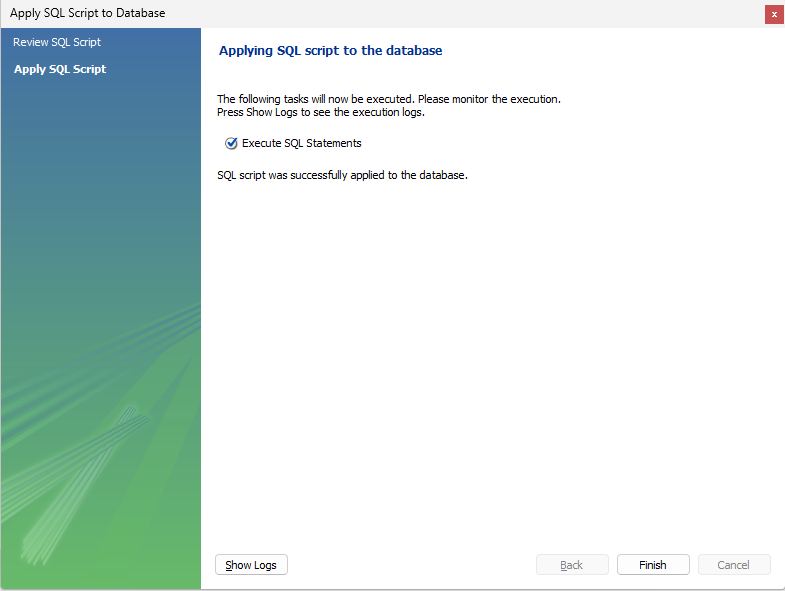


Figure Schema Creation Confirmation

After confirmation a new empty database with our specified name will be created.

A screenshot of a computer

AI-generated content may be incorrect.

Figure Empty Database Creation Success

1. **Generated Database dump Json file.**

A screenshot of a computer

AI-generated content may be incorrect.

Figure Django Directory Structure Before Data dump

After successfully setting up the MySQL container and creating an empty database through MySQL Workbench, the next step was to transfer the existing data from the development environment (SQLite) into the new MySQL database.

To accomplish this, I used Django’s built-in manage.py command to generate a JSON-formatted database dump from the current SQLite database:

“*python manage.py dumpdata > db.json*”

This command exports all data from the existing database into a file named db.json. The JSON format is compatible with Django’s loaddata command, making it easy to import the data into any new database, in this case, MySQL.

Using a JSON dump is a highly effective way to preserve the data integrity and maintain relationships between models while switching from one database system to another.

This step ensured that all application data, including user accounts, uploaded files, logs, and configurations, could be migrated safely to the new MySQL setup without manual re-entry or data loss.

A screenshot of a computer

AI-generated content may be incorrect.

Figure Directory Structure After Running Data dump Command

The database dump file is successfully created and its’s contents is given below:

A computer screen shot of a black screen

AI-generated content may be incorrect.

Figure Contents of db.json Dump File

1. **Updated the Databases configuration in settings.py to connect to the MySQL instance.**

After preparing the MySQL environment using Docker and generating the database dump, the next crucial step was to connect the Django application to the MySQL database by updating the DATABASES configuration in the settings.py file.

A computer screen with white text

AI-generated content may be incorrect.

Figure Default Database Configuration

By default, Django uses SQLite for development. To switch to MySQL, the following configuration was added in the settings.py file:

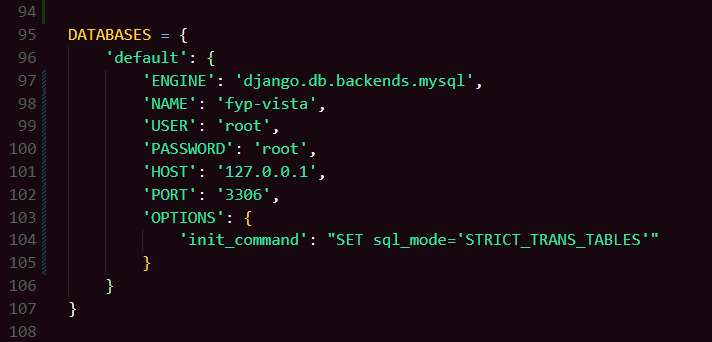


Figure Updated Databases Configurations

**ENGINE:** Specifies the database backend to use (mysql).

**NAME:** The name of the MySQL database created earlier.

**USER and PASSWORD:** Credentials for accessing the database (as defined in the Docker run command).

**HOST and PORT:** Point to the MySQL container running on the local machine via Docker.

**OPTIONS:** Ensures compatibility by setting strict SQL modes during initialization.

1. **Ran Django’s makemigrations and migrate commands to recreate tables in the MySQL database.**

To test if the Django web application can connect to the MySQL database successfully, I ran the Django web application, and it ran successfully indicating a connection establishment with the database.

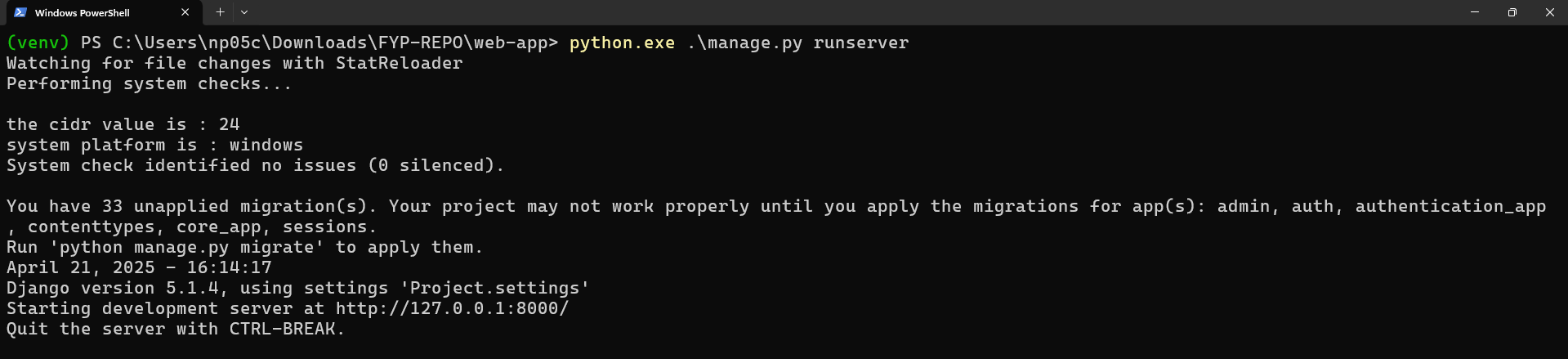


Figure Running Django Application

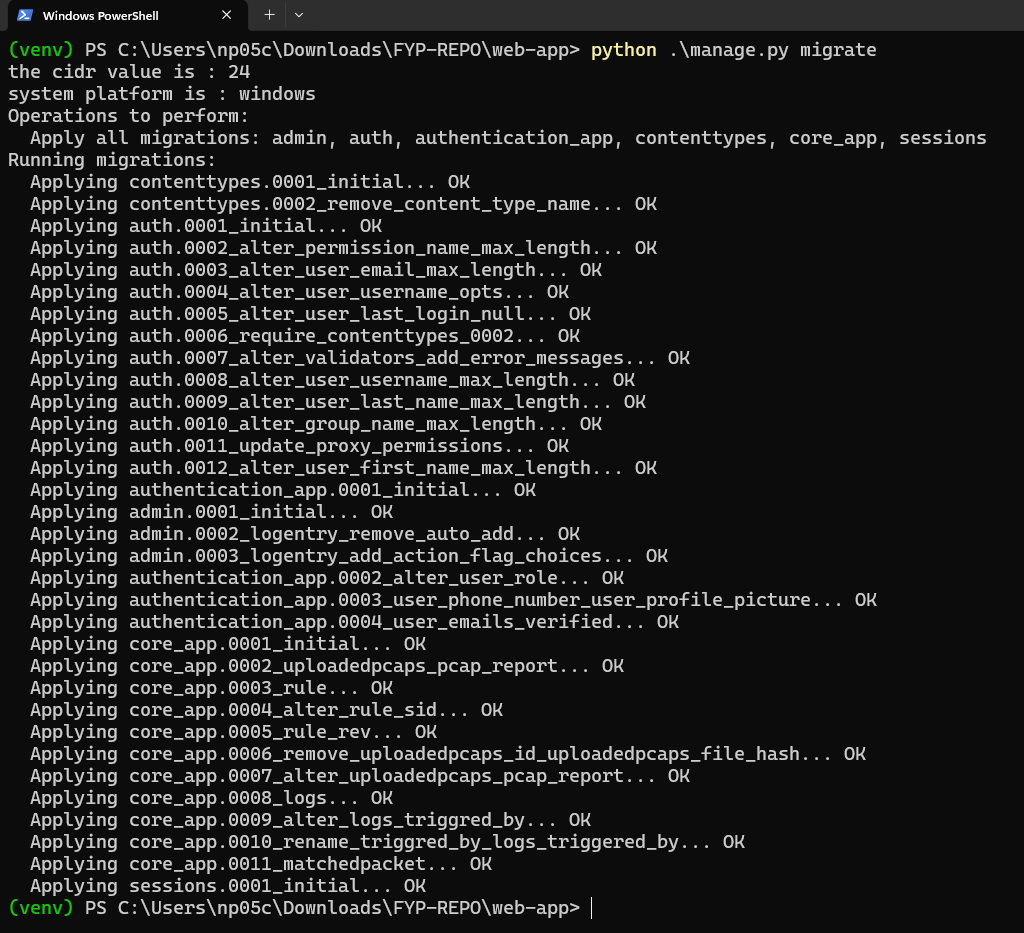


Figure Migration Result

The above figure displays the result of running the migration commands, confirming that Django applied all necessary changes without error.

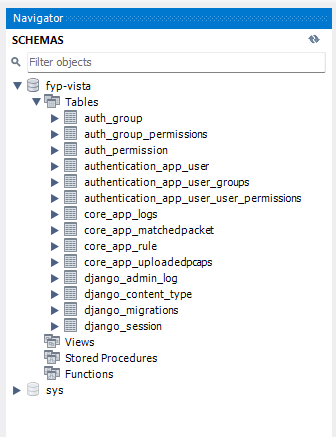


Figure Created Tables in Database

The figure above shows us that the tables were created successfully after applying migrations in the terminal.

1. **Import the Database dump Json file.**

Once the MySQL database was properly connected and the required tables were recreated using Django migrations, the next step was to import the previously exported database dump back into the system. Django allows importing data using the loaddata management command. The dump file generated earlier using dumpdata is in JSON format, containing serialized representations of the data from the SQLite database.

The command used to import the dump file was:

*“python manage.py loaddata db.json”*

* db.json is the name of the file generated during the export process.
* This command reads the JSON file and populates the corresponding tables in the newly connected MySQL database.

Before running this command, it’s important to ensure:

* The tables were created by running makemigrations and migrate.
* There are no conflicting entries (such as duplicate primary keys).
* The MySQL server is running and accessible.

After running this command successfully, all existing records were restored into the MySQL database, ensuring a seamless transition from SQLite to MySQL without data loss.

A screenshot of a computer

AI-generated content may be incorrect.

Figure Result of Loading Data

1. **Verify the data in the MySQL workbench.**

After successfully importing the db.json data into the MySQL database, the next step was to visually verify the data to ensure that the migration from SQLite to MySQL was successful and consistent.

To do this, I used MySQL Workbench, a graphical user interface that allows developers to manage and inspect MySQL databases efficiently.

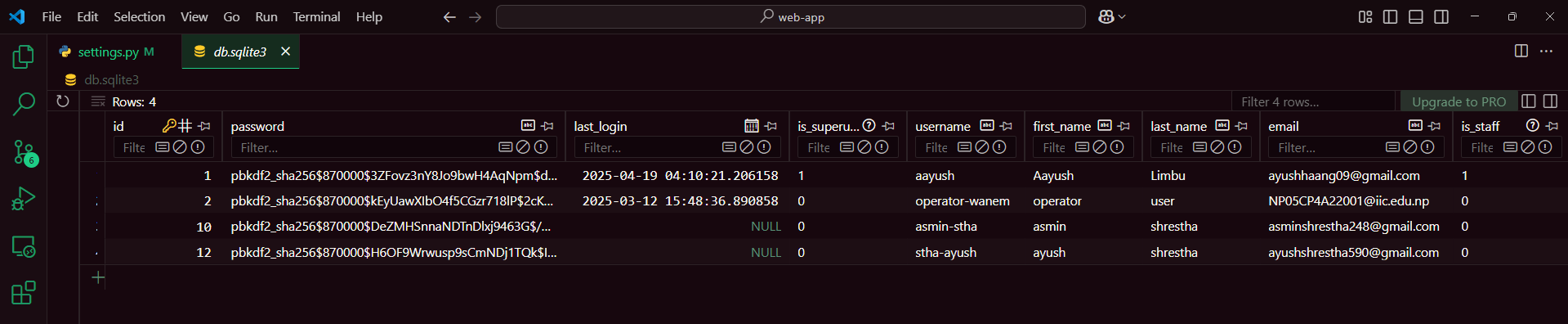


Figure User Data in SQLite Database

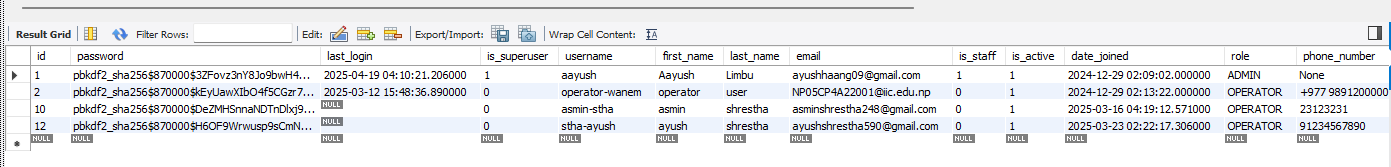


Figure User Data in MySQL Database

Figure 36 shows the User data as it was originally stored in the SQLite database before migration.

Figure 37 displays the same User data now appearing in the MySQL database after the transition.

This side-by-side comparison confirms that:

* All records were correctly imported.
* The table structure and relationships were preserved.
* Data integrity was maintained throughout the transition.

### Web Application

The web application serves as the frontend interface and backend logic of the system, it is developed using the Django web framework (Python-based). Django was chosen for its robust security features and rapid development capabilities among many other benefits it provides for web development.

The web application interacts directly with the MySQL database, handling CRUD operations through Django’s ORM (Object-Relational Mapping). This ensures seamless communication between the application and database layer without writing raw SQL queries.

#### Application Separation

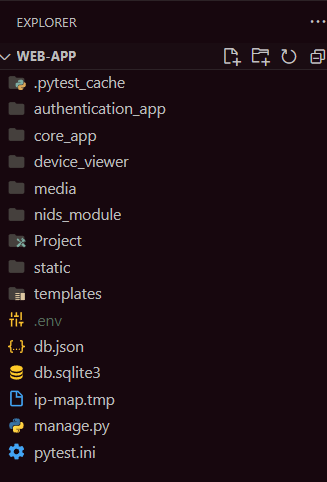


Figure Web App Structure Folder

For ease of development and maintainability in the future when this project needs to be updated or upgraded, it is split into different apps which have specific functionality. The Django applications of this project is:

* 1. authentication\_app
  2. core\_app
  3. device\_viewer

#### Web Application Endpoints

URL endpoints of Project folder:



Figure Urls.py of Project folder

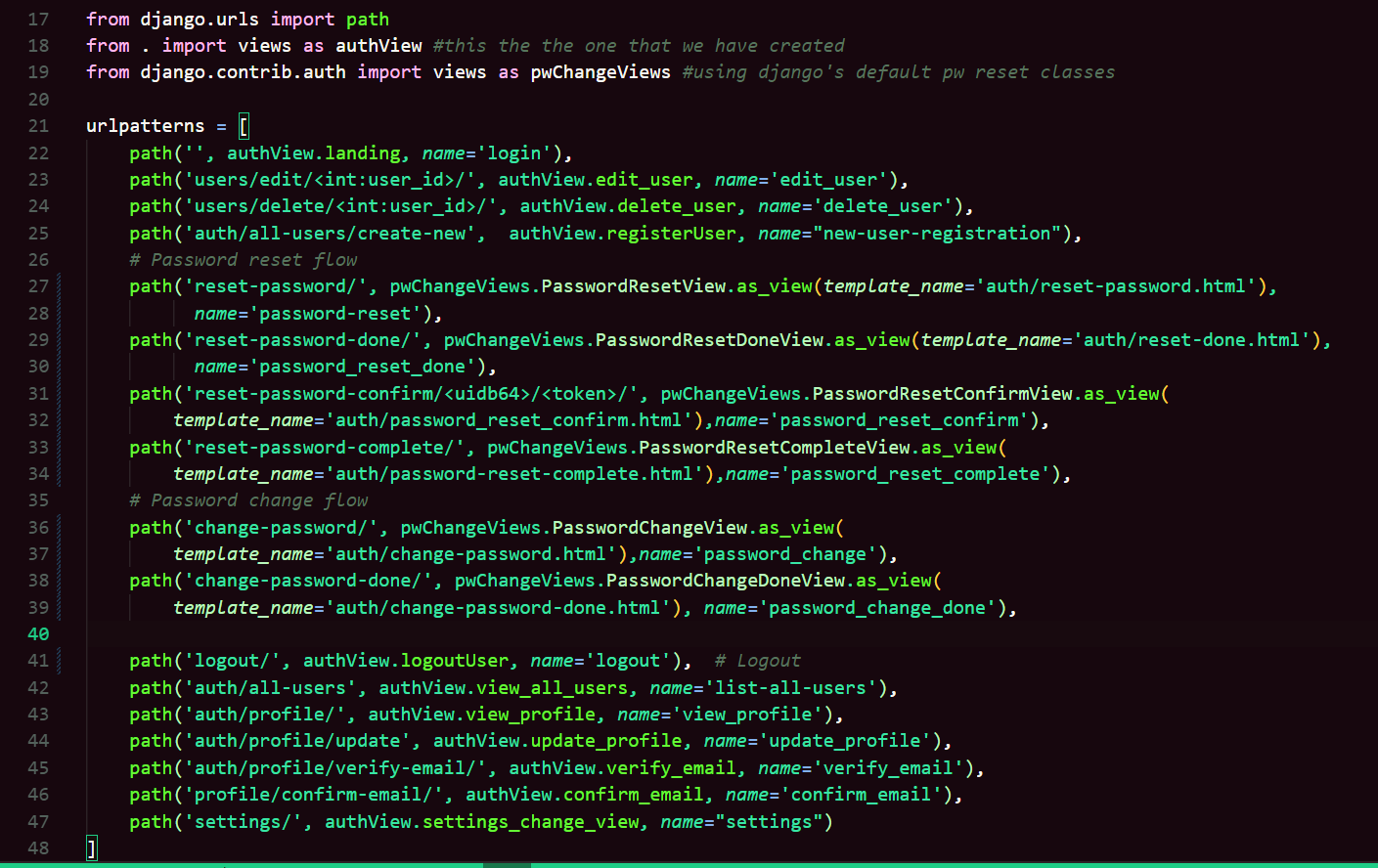


Figure Urls.py of authentication\_app

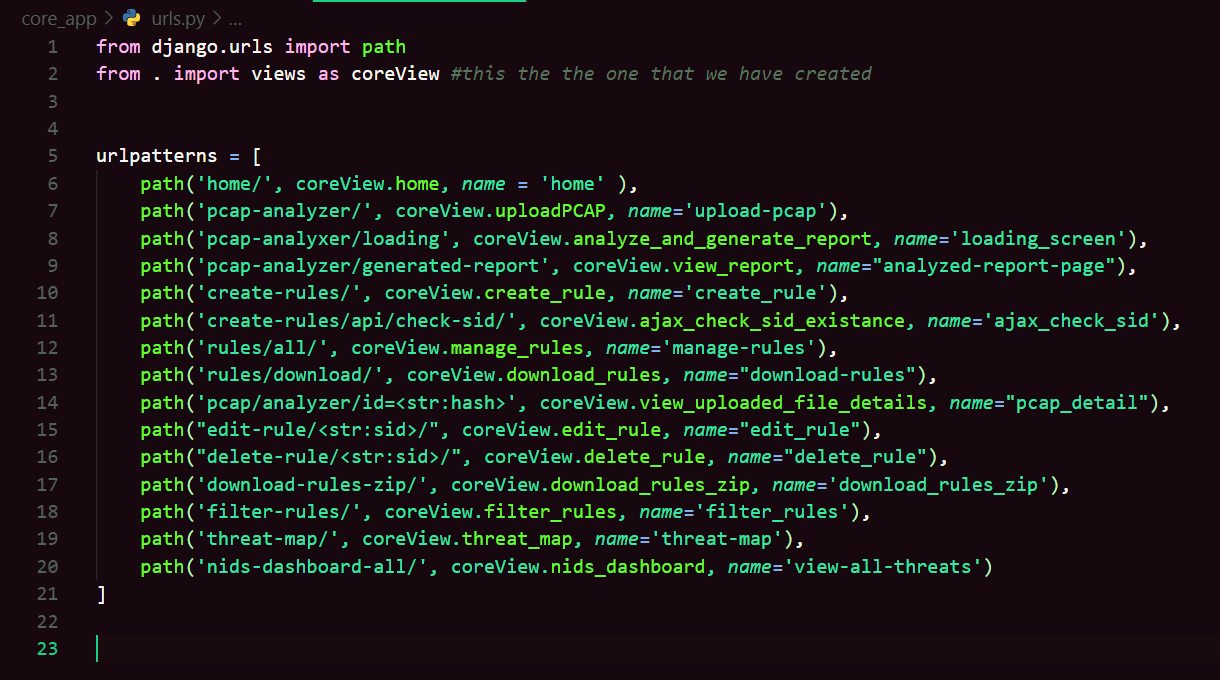


Figure Urls.py of core\_app

A computer screen with text

AI-generated content may be incorrect.

Figure Urls.py of device\_viewer app

# Testing and Analysis

## TEST PLAN

### UNIT TESTING, TEST PLAN

### SYSTEM TESTING, TEST PLAN

## Unit Testing

## System Testing

System testing is a crucial phase of software testing and development process, which ensures the validation of a fully integrated system or a software product. It involves evaluating the complete system's specifications and functionalities, considering that the software is just one component of a larger computer-based system. The main goal of this is to perform a series of diverse tests that thoroughly exercises the entire computer-based system, it includes interaction with other software and hardware components. By performing system tests, the overall functionality of the system can be assessed and validated. (Hamilton, 2024)

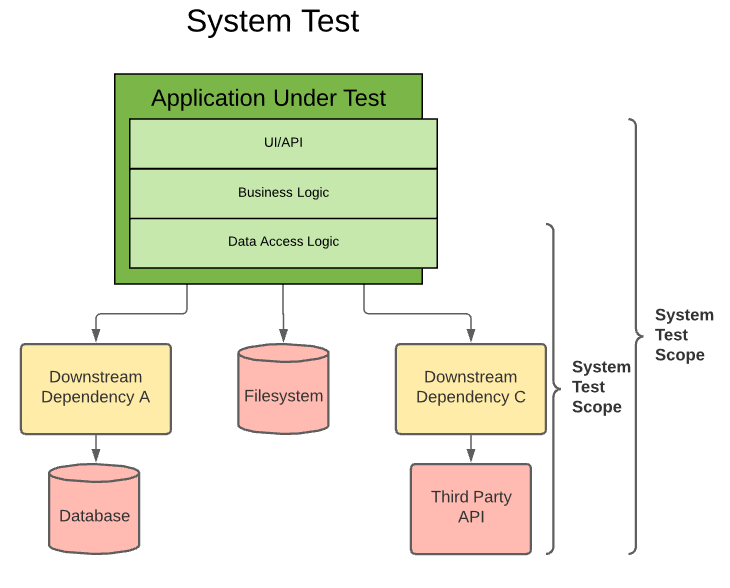


Figure 43 System Testing

Test 1: Starting the Django Development Server

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## CRITICAL ANALYSIS

# CONCLUSION

## LEGAL, SOCIAL AND ETHICAL ISSUES

### LEGAL ISSUES

### SOCIAL ISSUES

### ETHICAL ISSUES

## ADVANTAGES

## LIMITATIONS

## FUTURE WORK

# References

Hamilton, T., 2024. *www.guru99.com.* [Online]   
Available at: https://www.guru99.com/system-testing.html  
[Accessed 15 04 2025].

# BIBLIOGRAPHY

# APPENDIX

## APPENDIX A: PRE-SURVEY

### PRE-SURVEY FORM

### SAMPLE OF FILLED PRE-SURVEY FORMS

### PRE-SURVEY RESULT

## APPENDIX B: POST-SURVEY

### POST-SURVEY FORM

### SAMPLE OF FILLED POST-SURVEY FORMS

### POST-SURVEY RESULT

## APPENDIX C: SAMPLE CODES

### SAMPLE CODE OF THE UI

### SAMPLE CODE FOR THE AUTOMATION SCRIPT

## APPENDIX D: DESIGNS

### GANTT CHART

### WORK BREAKDOWN STRUCTURE

### ALGORITHMS & FLOWCHARTS

### 3D MODELLING

### HARDWARE ARCHITECTURE

### DATA FLOW DIAGRAMS

### USE CASE

### WIREFRAME

## APPENDIX E: SCREENSHOTS OF THE SYSTEM

## APPENDIX F: USER FEEDBACK

### USER FEEDBACK FORM

### SAMPLE OF FILLED USER FEEDBACK FORMS

## APPENDIX G: FUTURE WORK

### READINGS FOR FUTURE WORK