Bachelor Thesis

Computer Science and Engineering 300 credits

Improving Data Management in a Stamp Business through a Web-based interface

Computer Science and Engineering 15 credits

Halmstad September 1, 2025

Victor Nordblad

Elias Cregård

HALMSTAD UNIVERSITY

Abstract

This thesis addresses the challenges faced by a stamp-selling company in managing extensive data manually through Excel files, which often led to errors and inconveniences, such as incorrectly input data and time-consuming manual data management. To solve this, a web application was developed using ReactJS for its GUI, Flask for backend development, and Google Cloud SQL for scalable storage. The system integrates real-time WooCommerce synchronisation via webhooks and enforces input validation to prevent inconsistencies. Performance tests demonstrated that the system handles CRUD *Create, Read, Update, Delete* operations efficiently, being able to fetch 2000 stamps in under 200ms, ensuring real-time usability for inventory management and preventing workflow disruptions. Security tests confirmed resilience against attacks such as SQL injections. By replacing Excel sheets with a web-based application, the project not only improves data reliability and performance but also lays the foundation for future scalability and integration with additional business tools.

Keywords Web Application, ReactJS, TypeScript, Python, Flask, Database, MySQL, SQLAlchemy, CRUD operations, REST API, Google Cloud

Sammanfattning

Detta examensarbete behandlar de utmaningar som ett frimärkesföretag stod inför vid hantering av stora datamängder manuellt genom Excel-filer, vilket ofta ledde till fel och ineffektivitet, såsom felinmatad data och tidsfördrivande manuel datahantering. För att lösa detta utvecklades en säker och effektiv webbapplikation med ReactJS för ett användargränssnitt, Flask för utveckla ett backend-system och Google Cloud SQL för en skalbar datalagring. Systemet integrerar realtidssynkronisering med WooCommerce via webhooks och tillämpar validering av indata för att förhindra inkonsekvent data. Prestandatester demonstrerade att systemet hanterar CRUD-operationer effektivt och kan hämta upp mot 2000 frimärken på under 200 millisekunder, vilket säkerställer realtidsanvändning för lagerhantering. Säkerhetstester bekräftade skydd mot attacker som SQL-injektioner. Genom att ersätta Excel-ark med en web-baserad applikation förbättras inte bara datakvaliten och prestandan, utan även möjligheterna till framtida skalbarhet och integration med andra affärsverktyg.

Nyckelord Webbapplikation, ReactJS, TypeScript, Python, Flask, Databas, MySQL, SQLAlchemy, CRUD-operationer, REST API, Google Cloud

Preface

This work was done in collaboration with the company Star and Stamp Store. We are especially grateful to the company's owner, whose support, availability, and practical insights were invaluable throughout the entire project. We would also like to give a special thanks to our supervisor, Wagner Ourique De Morais, for his continued guidance, encouragement, and genuine interest in our work.

Contents

Abstract . . . . . **iii**
Sammanfattning . . . . . **v**
Preface . . . . . **vii**
Contents . . . . . **ix**
List of Figures . . . . . **xi**
List of Tables . . . . . **xiii**
Listings . . . . . **xv**
1 Introduction **1**
1.1 Background 1
1.2 Problem Statement 1
1.3 Purpose and goals 2
1.4 Requirements 2
1.5 Related Work 3
2 Technical Background **5**
2.1 Web Application 5
2.1.1 Three-Tier Architecture 5
2.2 Relational Database 7
2.3 Web Framework 7
2.3.1 Framework Types 7
2.3.2 Object Relational Mapping 8
2.4 Webhook 10
3 Method **11**
3.1 Development Process 11
3.2 Technical Design 12
3.2.1 Frontend 13
3.2.2 Backend & Data Layer 14
3.2.3 Deployment 20
3.2.4 Testing 21
3.2.5 Development environment and Collaboration tools 23
3.2.6 Deployment 24
4 Results **25**
4.1 Functional Test Results 25
4.1.1 CRUD Operations 25
4.1.2 Authentication 26

4.1.3 Excel Generation 26
4.1.4 Webhook Synchronization 27
4.2 Non-Functional Test Results 27
4.2.1 Data Integrity and Consistency 27
4.2.2 Security 27
4.2.3 CRUD Operation 27
5 Discussion 31
5.1 Analysis of functional results 31
5.1.1 CRUD Operations 31
5.1.2 Authentication 32
5.1.3 Webhook Synchronization 32
5.2 Analysis of non-functional results 32
5.2.1 CRUD Operations performance 32
5.2.2 Security 33
5.2.3 Database Design and Normalisation 33
5.3 Development Process Reflection 34
5.4 Societal Aspects 34
6 Conclusion 35
6.1 Future Work 35
Bibliography 37
A GUI Images 39

List of Figures

2.1 Simple graphical explanation of a three-tier architecture 6

3.1 Architecture Diagram 12
3.2 Final ERD 16
3.3 Insert stamp workflow diagram 18

A.1 The login page users are directed to when entering the web application. 39
A.2 The stamp list page where users can alter, delete and view stamps. 40
A.3 The page where users can insert new stamps, with an example of how the drop-down inputs look. 41
A.4 The page where users can generate Excel sheets with information about the stamps from the database. 42
A.5 The page where the admin can manage users in the system. 42

List of Tables

1.1 Comparison of related work and implementation choices 4

3.1 Defined KPI's used during testing 21

4.1 Read operation measurement with ~30,000 stamps 28
4.2 Read operation measurement with ~100,000 stamps 28
4.3 Filtering functionality (subqueries) 28
4.4 Create, Delete and Update operation measurements with ~30,000 stamps 29
4.5 Create, Delete and Update operation measurements with ~100,000 stamps 29

Listings

2.1 Raw SQL Example 9
2.2 Object Relational Mapping Example 9

3.1 An endpoint for handling user log in 19

A.1 An example API endpoint that handles HTTP requests 41
A.2 Data migration Excel Sheet column handler 43
A.3 An example of how to authenticate webhooks 43
A.4 An example of a webhook endpoint 44
A.5 Dockerfile for building and running the backend 44
A.6 The requirements.txt file for the backend application 45
A.7 Dockerfile for building and running the frontend 45

Chapter 1

Introduction

1.1 Background

This thesis was done together with a company that works with selling stamps. Currently, they manage all of the information about the stamps that they are selling in an Excel file. This is not a suitable solution for them, as they find it difficult to navigate and manage. It is common for them to input incorrect data or forget to input it all together. Therefore they need a better way to manage their data so they can eliminate these issues. They have requested a data management system for managing their stamps hosted on the cloud-based hosting platform Google Cloud, consisting of a *graphical user interface* (GUI) connected to a database to be developed in such a way so that they can input stamps and avoid the aforementioned issues. They have also requested that this system shall be synchronized with their webstore and update the content of the database, such as the price and availability of the stamps. This will be a web-based application as this approach offers great advantages for managing data. As opposed to having a program running on a user's device, running the application on the web requires no installation of any program and lets any device that has a web browser access the application, which improves accessibility. Additionally, having the application running on the web offers great advantages for scalability, as new functionality can be added to the website without having users download new versions of the application as opposed to the aforementioned program-based approach. Finally, it is vital for the application to be hosted on a server as it will have to support real-time data to be synchronized with the web store.

1.2 Problem Statement

The problem with file-based data storage solutions, such as Excel sheets, is that the structure of the data inside the files is dependent on the program that uses the data, meaning different programs can not access the same data as easily [1]. Furthermore, managing data manually using Excel sheets creates many opportunities for user

error. This is why a more streamlined way of managing data is needed. A web-based data management system with a GUI designed in a way that prohibits user error can fix these issues. However, there are other factors to consider when developing web-hosted data management applications. How will retrieval speeds, data integrity, and scalability improve when transitioning from a file-based Excel system to a web-based application, and what impact will these changes have on the overall system? What potential security issues can arise, and how can they be mitigated?

1.3 Purpose and goals

The purpose of this project is to develop a fully functional data management system that will solve the data management issues that come with storing data in an Excel file. These issues include time-consuming navigation and management as well as incorrectly inputted data.

The main goal is to create a solution that simplifies data handling and ensures data integrity and security. This means that every part of the system reflects the same data at any given time and protects the system from unauthorized access or malicious attacks. The system should support efficient Create, Read, Update, and Delete *CRUD* operations to the database. Although there's no universally agreed-upon benchmark for what defines 'efficient' CRUD operation performance in web applications, given the variability in database size, system infrastructure, and network speed, recent studies suggest that for datasets up to 50,000, CRUD operations should be executed within 0.5 seconds, and for larger datasets, up to 100,000, within 2 seconds [2]. The system should also support scalability to ensure future growth in the database, as well as security measures to ensure data integrity, meaning the stored information remains accurate, consistent, and reliable over time.

1.4 Requirements

- **Functional Requirements**
  - The web application should be able to authenticate users to ensure no unauthorised access.
  - The system should be able to store relevant stamp data that is used by the webstore, such as stamp categories, stamp price, etc.
  - The user interface should enable the user to view and manipulate the data in the database.
  - The system should validate user input before storing the data to ensure no missed inputs or faulty values are inserted into the database.
  - The system should support real-time data management to be able to synchronise with the webstore by changing the current price and sold status of stamps when they change on the webstore.

- **Non-Functional Requirements**
  - The system shall be secure against SQL injections.
  - The system should be able to execute operations such as create, update, and delete within a 0.5-second span with its current size, 30,000 stamp entries.
  - The system should be able to handle operations with over 100,000 stamps stored in the database within a 2-second span.
  - The system should be able to fetch and display up to 2000 stamps within 0.5 seconds.
  - The database should be hosted on Google Cloud SQL, and the web application should be hosted using Google Run.

1.5 Related Work

Several studies and articles have explored front- and back-end web application development, offering insights that are relevant to our project.

A library management system was developed using ReactJS, demonstrating the advantages of using ReactJS when constructing a web-based data management system [3]. ReactJS is emphasized as a powerful tool due to its use of reusable component-based structure, virtual document object model *(virtual DOM)*, which enhances rendering as well as its easy integration with back-end technologies. Its simplicity, developer-friendly environment, and broad community make it especially suitable for responsive, maintainable, and high-performance web applications.

A web application was constructed using Flask to manage and interact with a database [4]. Flask was chosen for its lightweight structure, flexibility, and ease of use, allowing for fast development. The project demonstrates how Flask supports web development and enables smooth handling of data operations such as insertion, updates and queries, making it a useful framework for building web applications synchronised with a database.

While Flask does support frontend web development through server-side rendering, its primary use case is for building backend logic. It is, therefore, not optimal for making functional user interfaces on the web. On the other hand, React's primary use case is for developing frontend user interfaces and is, therefore, better suited for developing a proper GUI [5].

While Flask offers advantages such as a fast learning curve and high flexibility, it is important to keep in mind its limitations in performance. A comparative study of frameworks has shown that Flask tends to underperform compared to other modern and performance-specific frameworks [6]. It was concluded that Flask is good at handling moderate traffic and requires optimisations and additional tools to match the other frameworks.

Article	Key Findings	Comparison
1	ReactJS is effective for data management systems due to reusable components, virtual DOM, and backend integration.	We also utilized ReactJS for GUI, leveraging its component-based structure and virtual DOM.
2	Flask is lightweight and flexible for backend development, enabling smooth database interactions.	We also utilized Flask for its simplicity and integration with SQLAlchemy.
3	React is better suited for GUI development compared to Flask	Our artefact mirrors this by combining React for the frontend and Flask for the backend.
4	Flask is simpler and more flexible for smaller/medium projects.	Our choice of flask aligns with this as our artefact is on the smaller side and required rapid development.

Table 1.1: Comparison of related work and implementation choices

The now-mentioned articles provide valuable insights that will help the development of this project. As mentioned, ReactJS is highly suitable for developing frontend web applications due to its reusable components, virtual DOM, and seamless backend integration. Its simplicity and strong community support make it an ideal choice for frontend development. On the backend, Flask has been shown to be effective for smaller applications. Its lightweight and flexible architecture supports rapid development and smooth handling of data operations.

Chapter 2

Technical Background

2.1 Web Application

Web applications are software applications that are created using web-based technologies, such as HTML and JavaScript. These applications are typically hosted on a web server and can be accessed through a web browser. This is an advantage over more traditional software applications, that requires the user to have the application installed on their device. This, in turn, makes the application more accessible. The application can also be updated without requiring the client to install a new update, which ensures that the application is always up to date [7].

2.1.1 Three-Tier Architecture

Web-based software applications are often based on multitier architectures. These architectures separate the application into different tiers that allow the different parts of the system to be updated individually without requiring the other tiers to be updated. One common multitier architecture is the three-tier architecture. This architecture separates the application into three tiers [8]:

- A **presentation tier** that consists of the user interface. This is the part that the user interacts with. In the context of a web application, this tier consists of HTML files and JavaScript code that is sent to and interacted with through the client's web browser. This part of the application is often referred to as the frontend.
- A **data tier** that handles data used by the application. It is often in the form of a file storage system or a database managed by a DBMS (see section 2.2).
- A **logic tier**, or the middle tier, that handles computation. This tier directly communicates with both the presentation layer and the data layer and exchanges data between them. It is often made in the form of an *Application Programming Interface* (API). It also handles user sessions and keeps track of data access permissions. This part of the application is often referred to as the backend.

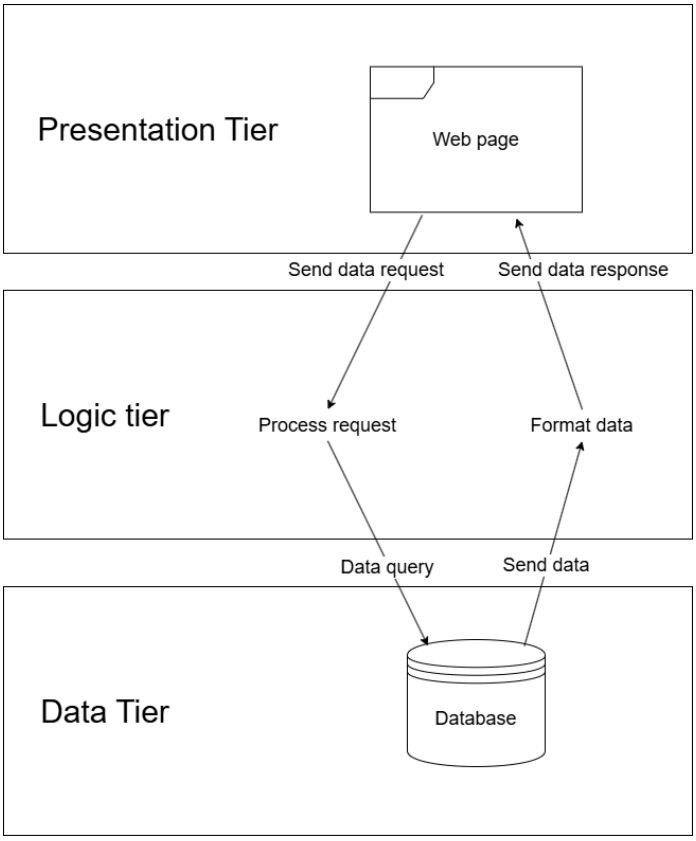

Figure 2.1: Simple graphical explanation of a three-tier architecture

The logic tier being an API brings many great advantages in the form of scalability. APIs offers easy communication with many different applications, and thus making it simple to add new module to the application down the applications lifecycle, promoting scalability.

2.2 Relational Database

A database is an organised structure of data that allows for efficient storage, retrieval, and management of information. In a relational database, data is grouped into tuples, or rows, that are then inserted into tables with predefined structures. Tuples can be identified by their *Primary Key*'s PK and tables can be related to eachother with *Foreign Key*'s (FK). A database is managed by using a *Database Management System* (DBMS). Together, a database and a DBMS, along with the applications that are associated with them, make a database system. When developing a database structure, metrics such as integrity, performance and scalability are crucial to keep in mind.

A common approach when constructing a database is to follow normalisation rules. Following these rules reduces redundancy and increases integrity in the database. *First Normal Form* (1NF) is the first rule and restricts each tuple to only containing one atomic value per row, and each record is unique. *Second Normal Form* (2NF) builds on 1NF by removing partial dependencies and ensuring that all non-key attributes must depend on the whole primary key. *Third Normal Form* (3NF) builds on 2NF by removing transitive dependencies and ensures that non-key attributes must depend only on the primary key in the table.

2.3 Web Framework

Web Framework's are software tools that help develop web applications. They provide core functionality for features that are often required when building web applications, such as user session management and database interfacing, among other things.

2.3.1 Framework Types

There are two main types of web frameworks, client-side frameworks, and server-side frameworks.

Client-Side

Client side frameworks are used for developing applications where most computations are performed by the client, in their browser. These are used for developing *Single-Page Applications* (SPA), where a single page is fetched from the application server, and then manipulated by the client for displaying new information to the user. This offloads computation from the application server and instead rely on the client to perform the computation. SPAs are beneficial when user responsiveness is important, i.e, for user interfaces. Therefore they are suited for the presentation tier as mentioned in section 2.1.1.

Server-Side

Server-side frameworks, on the other hand are used for developing applications where most computations and page generations are performed on the application server, which offloads the computational load of the client. This comes with the downside that web pages have to be refreshed by the client to reflect changes and thus leads to a less responsive user interface. Therefore they are not best suited for creating user interfaces. This is better suited for applications where the focus is more on computation, such as web APIs. This is useful for developing the logic layer, as mentioned in section 2.1.1, as there is no need responsive user interaction.

Two popular server-sise frameworks are Flask and Django. Both frameworks offer basic web frameworks functionality such as handling and responding to incomming web requests from multiple clients. While Django offer loads of other functionality related to server-side applications, like user authentication and authorization, an ORM (see section 2.3.2), and a unit test framework among other things. Flask on the other hand task a different approach. Flasks enables the developers to implement features in any way they wish, one at a time. This leads to a much more lightweight framework that performs better than Django for smaller projects [9]. There are packages made to be used with Flask if the developer wishes. For example, Flask-Login, which is a user authentication and log-in session handling [10]. There is also a version of the ORM SQLAlchemy called Flask-SQLAlchemy (see more in section 2.3.2).

2.3.2 Object Relational Mapping

Object Relational Mapping (ORM) is a way to communicate with relational databases using programming code. ORMs create an abstraction layer between an *Object Oriented Programming* (OOP) language and a DBMS to allow for normal database data manipulation using programming language syntax instead of having to write SQL queries. ORMs are very often used in the logic tier of a web application to communicate with the data tier (see section 2.1.1).

In a relational database, as mentioned in section 2.2, data is grouped into tuples, or rows, that are then inserted into tables with a predefined structure. ORMs manages this by grouping the data into objects. For example, if in a relational database, there is a table called *Cars*, an ORM would handle every tuple in this table as Cars objects. Below is an example of how simple read and write operations from this table would look using raw SQL versus using an ORM, in this case, SQLAlchemy, in the programming language Python. Both examples tries to get the make of the car with an id of 5, and change the color of the car to blue.

Listing 2.1: Raw SQL Example

```
query = "SELECT make FROM cars WHERE id = 5;"
car_make = cursor.execute(query).fetchone()[0]
query = "UPDATE cars SET color = "Blue" WHERE id = 5"
cursor.execute(query)
```

Listing 2.2: Object Relational Mapping Example

```
car_object = Cars.get(5)
car_make = car_object.make
car_object.color = "Blue"
```

Listing 2.2 shows that a Cars object is returned when the *get* method is called on the *Cars* class. The class acts as the Cars table in this case, and the method returns the object using the primary key specified by the method's argument. The attributes of the object, like the make in this case, can easily be accessed. As seen in listing 2.1, the query only returns the value of the specified field with no connection to the actual database entry. Furthermore, this example also shows that data can easily be manipulated by manipulating the object itself, as seen in listing 2.2, as opposed to in listing 2.1, where a new query has to be created to perform the new operation.

Another advantage of using ORMs over manually writing SQL queries is that ORMs help ensure safe database code by sanitising inputs. Sanitising inputs means treating external data (user input, request payloads, etc.) as values only, not as part of SQL syntax. ORMs achieve this by using parameterised (prepared) statements or bound parameters, automatic escaping, and type checking so user data cannot change the structure of a query and inject malicious SQL. This reduces the risk of SQL injection [11] and complements other measures such as input validation.

PHP, Python and Java are examples of OOP languages that are supported by ORM tools. As the API Server will be developed using Python, an ORM tool for Python will be chosen for this project. There are many ORM tools available for Python. Some examples are Django, SQLAlchemy, SQLObject and Web2py. As Flask will be used to develop the backend of the application, choosing an ORM tool that is compatible with Flask will facilitate seamless integration and simplify the implementation process. SQLAlchemy has a version specifically made to be used with Flask called Flask-SQLAlchemy. This package simplifies the use of SQLAlchemy together with Flask by letting the connection to the database be handled and ran inside of the Flask application server [12].

2.4 Webhook

A webhook is an HTTP-based callback mechanism that enables real-time communication between applications. Instead of repeatedly polling a server for updates, a client can register a unique URL (webhook endpoint) and specify what events it wants to be notified about. When the specified event occurs, the server sends an HTTP POST request to the client's endpoint, containing a payload with the data relevant to the event.

A typical webhook request consists of a header that includes metadata such as content type and/or authentication tokens, which can be used to authenticate the POST request, and a body that carries the event data in formats such as JSON or XML. By eliminating the need for polling, webhooks reduce system traffic and improve system responsiveness. This minimizes the performance costs of adding more modules to a web application.

Using webhooks instead of polling introduces several challenges, primarily related to data delivery and security. Since the responsibility for ensuring successful delivery is on the sender, interruptions such as downtime on the receiving server may result in lost data, depending on how retries are configured on the sender's side. Furthermore, webhooks require the receiver to expose a publicly accessible endpoint to accept incoming requests. This can create potential security risks, as unauthorised parties could attempt to send malicious data to that endpoint, and may succeed if proper validation and security measures are not in place.

Chapter 3

Method

3.1 Development Process

During this project, the waterfall process was used to achieve a functional and finished product. The reason for using a waterfall-like approach, as opposed to an agile-like approach, is that the project had a fixed timeline and the requirements would not change during the development of the artefact. This project involved five development stages: a requirement analysis, design, implementation, testing, and deployment phase. During the requirement analysis phase, interviews with the company were conducted to identify their needs and problems with their Excel-based system, which informed the functional and non-functional requirements.

The design phase converted these requirements into technical specifications. A normalised database structure was developed using an Entity Relationship Diagram *ERD*, which gave a clear and structured way to represent the database and its relationships. This phase also included setting up the tools and technologies used during the development of this project, such as installing required development frameworks, setting up GitHub for collaborative development and configuring Google Cloud Run.

After the design phase, the implementation phase began. This phase included constructing the frontend, backend, implementing the database into Google Cloud SQL and migrating data from the Excel sheet to the constructed database. The database was constructed using Python using the Flask SQLAlchemy Object Relational Model (*ORM*) and implemented in the backend code. Data was migrated from the Excel sheet using a Python library called pandas, which comes with built-in features that make reading and migrating the Excel sheet seamless. The frontend was designed and developed using ReactJS, and was designed to fulfil the company's requirements. React also lets us create modular components that could be reused, such as our dropdowns, which made the interface development seamless. The backend was constructed using Python and Flask. Flask was used to create API endpoints that connected the frontend to the database. Each endpoint handles specific tasks such as */stamps* (see listing A.1) that both fetches already existing stamps as well as inserts new ones into the database. Synchronisation

with WooCommerce was also done through webhooks. SQLAlchemy was used for database operations, and Flask-Login was used for authentication.

Next was the testing phase. This included testing each functional and non-functional requirement to ensure the criteria had been fulfilled. Unit tests validated authentication, which included testing valid and invalid user credentials. Security tests conducted using a tool called SQLmap confirmed protection against SQL injection attacks [13]. Integration tests verified WooCommerce synchronisation with the database by processing multiple test orders on a staging site. Performance benchmarks demonstrated the system's ability to handle an overloaded database by performing CRUD operations on the database with 100,000 different stamps. User feedback from the company confirmed the GUI's effectiveness as well as its user-friendliness.

3.2 Technical Design

The constructed system is structured into three core parts that work together to deliver full functionality: a ReactJS frontend for user interaction, a Flask backend handling business logic and data processing, and a Google Cloud SQL hosted database for secure storage.

As illustrated in Figure 3.1, these components follow a layered architecture where the frontend communicates with the backend via REST API calls, while the backend processes requests and manages database interaction. A key feature is the real-time WooCommerce synchronisation, where webhooks automatically synchronise online sales with the database.

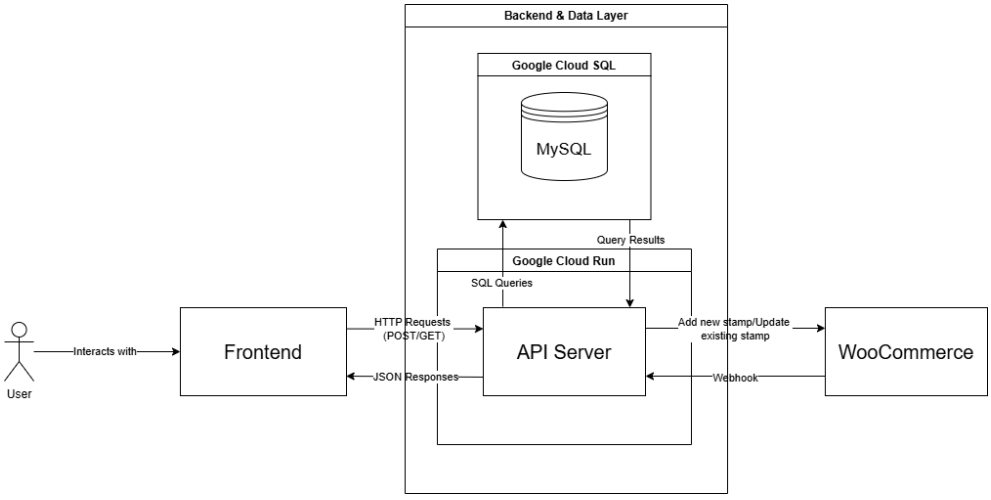

Figure 3.1: Architecture Diagram

3.2.1 Frontend

ReactJS was chosen as the GUI library for developing the web application because of its fast rendering process and wide range of built-in features that accelerate development . As one of the most widely used JavaScript libraries for building web applications, React offers extensive documentation and a strong community, which provides immense support throughout the development process. This choice aligns with the findings in the related work section, where ReactJS was highlighted for its efficiency, component-based design and its strong community (see Section 1.5) The constructed web application is divided into five main pages, beginning with the first page users see upon entry, the *Login Page*.

When designing the frontend, we applied several key design principles, including usability, input integrity, performance, and reusability. One key design decision was the use of dropdowns instead of free-text inputs for structured data like category, country, and collection. This ensured consistency in the database and minimized user input errors.

We also applied the DRY (Don't repeat yourself) principle by creating reusable React components for elements like dropdowns and stamp cards. This promoted consistent behaviour across the interface and made the code easier to maintain.

For performance, pagination was used to avoid rendering multiple thousands of stamps at once. This will lead to a more responsive interface, as no matter the size of the dataset used, the interface will only have to render a few stamps at once. React's built-in state management and Virtual-DOM allowed the application to respond efficiently to user input and data updates without requiring full page reloads.

These decisions were made to meet the system's requirements around data integrity and performance (see section 1.4).

The following paragraphs briefly describe how these principles were implemented across the system's main pages.

The login page is the entry point of the system and is responsible for handling user authentication (see A.1). It contains two input fields, one for the username and one for the password, both required for submission. When submitted, a POST request is sent to the backend with the entered credentials. If authentication is successful, the user is redirected to the main menu, and their session is stored securely using Flask-Login. In the event of invalid credentials, a generic message is displayed on the page and login is denied. To prevent brute-force attacks, login is blocked after enough tries.

To navigate between pages on the constructed web application, a simple sidebar with the different possible pages was implemented.

After logging in, users are directed to the stamp list page, which displays existing stamps in a scrollable and paginated list (see A.2). This list is fetched from the backend and rendered on the frontend with real-time sorting and filtering capabilities based on various stamp attributes. To optimize performance, a pagi-

nation system was implemented, only showing a limited number of stamps per page. Selecting a stamp opens a detail view, where users can view the stamps full information and access buttons to update or delete it. Updates lead the user to a prefilled form similar to the one used for inserting new stamps. Sorting options are also available to help users sort and quickly locate specific stamps.

The stamp insertion page is designed to help users add new stamps to the system efficiently while minimizing input errors (see A.3). The form combines free text fields and dynamically populated drop-downs to ensure consistency and data integrity. Attributes like country, category, and collection are selected using dropdowns that fetch the possible choices from the backend, allowing users to choose only from valid entries already stored in the database. Categories are handled using a recursive system, where selecting a parent category dynamically loads its child categories. The form also includes input validation to ensure that all required fields are completed before submission. Once submitted, the form sends the data to the backend via a POST request, where additional validation occurs before the stamp is added to the database.

The third page is where users can generate an Excel sheet containing all stamps in the database (see A.4). This page acts as a fallback mechanism in case the synchronization between the system and the WooCommerce webstore fails. The structure of the generated file matches the format the company previously used in Excel, ensuring compatibility with existing workflows. When the user requests an export, the backend begins generating the file using the Pandas library. In listing A.2, the class that is used to read the data in the Excel sheet is listed together with an example of how a column in the sheet is read. The frontend checks periodically for completion and is then automatically downloaded on the users computer.

The last page is the user handler page, which is a page only available to administrative users and allows for adding and removing users from the system (see A.5). To create a new account, the admin enters a username and password, which are sent to the backend and stored after hashing the password using bcrypt. All users are displayed in a list, and each entry includes a delete button that allows the admin to remove users directly. Any modifications immediately update the database and are reflected in the list. This page ensures that access to the system remains controlled and that new users can be added or removed without complication.

3.2.2 Backend & Data Layer

The *Backend & Data Layer* forms the foundation of the web application constructed during this project, enabling secure data storage, retrieval, and processing. It is implemented using Python and Flask. The backend consists of two key modules, including the API server and the database layer. These modules work together to handle requests from the frontend, process received data, and store it. The backend is hosted on Google Cloud Run, while the database is hosted on Google Cloud SQL, ensuring scalability and seamless integration.

Data Layer

The database was implemented using SQLAlchemy together with MySQL and hosted on Google Cloud SQL. Out of the three DBMSs that Google Cloud SQL supports, MySQL, PostgreSQL, and Microsoft SQL Server, MySQL was chosen due to our previous experience with it and its widespread popularity. Google Cloud SQL was chosen due to it being a request from the company, and Google Cloud SQL, together with SQLAlchemy, allowed for a seamless connection between the backend and database.

The database was designed using relational modelling principles and normalisation rules to ensure data consistency, eliminate redundancy, and support efficient querying. By organising data into logically separated tables with well-defined relationships, the system maintains integrity and scalability as the metadata in the database grows.

The ERD shown below illustrates the database structure that was implemented in Google Cloud SQL. It was designed to be able to store all essential information about a stamp while enabling efficient synchronisation between the internal database and the WooCommerce e-store. To avoid creating separate tables for each main and subcategory, a recursive structure was implemented in the category table. This allows both main and subcategories to be stored in the same table, where subcategories reference their parent categories through a **Parent_ID** attribute. Bridging entities such as the *CategoryAssignments* table are used to manage many-to-many relationships in the database. For example, a single stamp can belong to multiple categories, and a single category can include multiple stamps. The *CategoryAssignments* table links stamps to categories by storing the **stamp_id** and **category_id** as foreign keys, ensuring data integrity and enabling efficient queries.

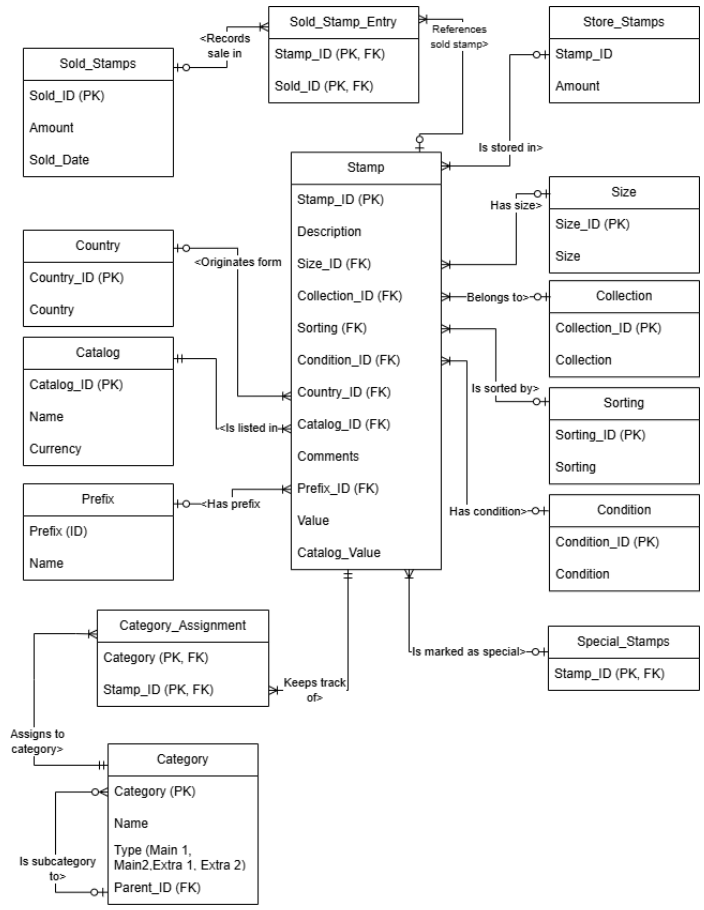

Figure 3.2: Final ERD

To allow for communication between the backend and the database, SQLAlchemy was utilised. SQLAlchemy is a powerful ORM which was chosen due to its ability to simplify database interaction by allowing developers to work with Python objects instead of writing raw SQL queries. Using SQLAlchemy, table models were defined in Python to represent the structure and relationships of the actual database. These models allow developers to perform Mysql queries and operations, such as inserting, updating, or deleting data directly through Python code. SQLAlchemy handles the translation of these operations by translating them into raw SQL queries, which are then executed in the database.

Backend

The backend of the web application was developed using Python and Flask. Python was chosen due to our previous knowledge with the programming language. Flask was chosen for its simplicity and extensive extension availability, which allowed for full customisation, rapid development, and seamless integration with other

components. As highlighted in (Section 1.5), Flask is frequently recommended for lightweight applications, which corresponds well to the needs of the constructed system. As one of the most widely used Python frameworks for building web applications, Flask offers comprehensive online documentation and a strong developer community, which provides valuable support during development.

The API server acts as the central hub of the backend, handling communication between the frontend and the database. It is responsible for handling incoming HTTP requests from the frontend, processing the data, interacting with the database, and returning appropriate responses. By organizing the API server into distinct modules such as *login handler*, *request handler*, and *webhook handler*, each focused on specific functionality, the system ensures maintainability, scalability, and separation of concerns.

The API server utilises Flask's built-in routing system to define endpoints for various operations, such as CRUD operations and user authentication. Each endpoint is associated with a specific HTTP method (e.g. **GET**, **POST**, **DELETE**) and a corresponding function that processes the request.

Below is an example of how inserting a stamp into the database works. When a user wants to insert a new stamp, a **POST** request is sent to the backend with the stamp information. This request is routed to a Flask endpoint called */stamps* (see listing A.1), designed to handle both **POST** and **GET** requests. Upon receiving the request, the backend first checks the HTTP method to determine the type of operation. Then it checks that the user is validated and has access to the requested data. Since it is a **POST** request, the backend proceeds to validate and process the received data.

The data is extracted from the request's body using Flask's *request.get_json()* method, which parses the JSON payload into a Python dictionary. This dictionary is then mapped to the database schema using SQLAlchemy's ORM, which translates Python classes into relational tables. For instance, the *Stamps(db.Model)* class defines attributes such as *country* and *size*, corresponding to database columns. This integration makes creating new entries a seamless operation. To enforce data integrity, a self-written method, *Stamps.make_entry(stamp_data)*, processes the parsed data and formats the input to match the database schema before creating a new *Stamp* object.

Once a new stamp is created, it is inserted into the database session using SQLAlchemy's *database.session.add()* method. This method prepares the database session for a new insertion. Finally, the changes are committed to the database using *database.session.commit()*, which executes the SQL query to insert the new stamp into the Mysql database hosted on Google Cloud SQL. If any failures occur during this process, such as validation or database connection failures, they are caught and returned as error responses. This workflow ensures that the data is validated, securely stored, and synchronised with the database.

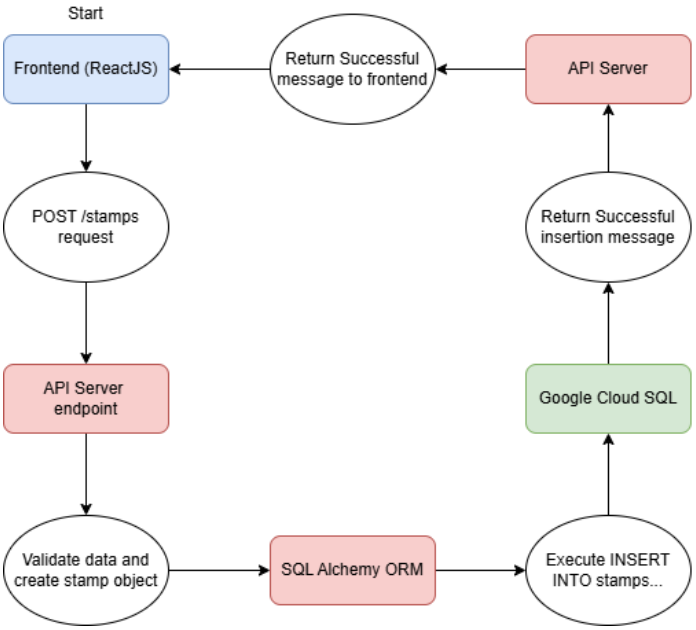

Figure 3.3: Insert stamp workflow diagram

User authentication is a critical component of the backend, ensuring that only authorized users can access the application. Flask-Login, a library designed to integrate seamlessly with Flask, was used to handle user authentication and session management. This library simplifies the process of managing user sessions by providing built-in support for features such as login, logout, and access control. When a user attempts to log in, their credentials are sent to an endpoint in the backend called */login*. Their credentials is then verified by hashing the submitted password using bcrypt and then compared with the stored hash in a table called *Users* in the database. If the credentials are valid, Flask-Login creates a session for the user, allowing them to access protected routes throughout the application. In listing 3.1, the method for login in a user to the application is listed.

Listing 3.1: An endpoint for handling user log in

```
from flask import request, jsonify, Response

@app.route("/login", methods=["POST"])
@login_required
def login() -> Response:
    try:
        data: dict[str, str] = request.get_json()
        password = data.get("password")
        username = data.get("username")
    except Exception as e:
        return jsonify({"error": str(e)}), 400
    user = User.get(username)
    if not user:
        return jsonify({"error": "User not found"}), 404
    user.authenticate_user(password)
    if not user.is_authenticated:
        return jsonify({"error": "Invalid password"}), 401
    if not login_user(user, remember=True):
        return jsonify({"error": "Login failed"}), 401
    return jsonify({"message": "Login successful"}), 200
```

Synchronisation between the backend and WooCommerce is essential to ensure that the database remains consistent with the e-store the company uses. This was achieved using WooCommerce webhooks, which send real-time notifications to the backend whenever specific events occur, such as when a new order has been created. When a webhook notification is received at the backend, it gets handled in the *webhook_handler* module and ends up in the */webhook/order_updated* endpoint (see listing A.4). The first thing the *order_updated()* function does is to authenticate the request received using a Hash-based Message Authentication Code *HMAC* signature (see listing A.3). This verification ensures that the request originates from WooCommerce and prevents unauthorised access. Upon successful authentication, the data is extracted from the webhook payload, including details such as sold stamp, their quantities and sold date. To maintain database synchronisation with the e-store and to ensure stamp quantities are updated only for finalised orders, the function checks the payload's 'status' attribute, proceeding only if the order received is marked as 'completed'. After checking the status, the inventory is updated in the *StoreStamps* table, and the sale is recorded in the *SoldStamps* and *SoldStampEntry* tables. This webhook-based synchronisation ensures real-time updates, reduces server load by eliminating the need for polling, and provides a secure and scalable solution for maintaining consistency between the database and WooCommerce e-store.

Security

Website security was a key concern during the development of the web application, as malicious attacks, such as stealing, manipulating, or removing data from the database, could compromise the integrity of the system. One of these attacks is SQL injection, where attackers exploit vulnerabilities in query handling to execute unauthorised SQL commands. To prevent SQL injections from happening, SQLAlchemy was used. SQLAlchemy ensures that SQL injections are mitigated by using parameterised queries, which safely handle user inputs by automatically escaping special characters and ensuring that user-provided data cannot alter the database. This approach effectively prevents SQL injection attacks and ensures that all database interactions are secure.

And as mentioned before, Flask-Login was also utilised to ensure that authorised users can access the web application and its functionality. To enhance security, the frontend includes the *withCredentials* option in any request made to the backend. This ensures that the cookies, such as the session cookie created by Flask-Login, are sent along with the request. The backend uses these cookies to verify the user's session and determine whether they are logged in. This mechanism adds an additional layer of security by preventing unauthorised access to protected routes. As you can see in A.1, there is a decorator (@login_required) that tells the backend to check that the user is verified before performing the request. If the user is not authorised, a response with the status code 401 that tells the client that the user is not authorised is returned.

To protect the system against brute force logins, the number of unsuccessful login attempts by users is saved. If too many unsuccessful login attempts have been made by the user, the user will be prohibited from making any more login attempts for a certain amount of time. In this case, 5 unsuccessful attempts will lead to the user being prohibited from making any more attempts until 1 minute has passed. If, after 1 minute has passed, the user makes another unsuccessful login attempt, the user will be locked out for 5 minutes. This timeout will stand for every successive unsuccessful login attempt. When the user finally makes a successful login attempt, the number of unsuccessful login attempts is cleared, and the user is granted access to the system. Furthermore, a minimum password length of six characters is required. This was chosen to counteract users from having insecure passwords, i.e. passwords that are easy to brute force.

3.2.3 Deployment

As per the company's request, the artefact would be hosted on the Google Cloud suite. When deploying the different parts of the artefact on Google Cloud, it was ensured that all the artefacts could communicate with each other by manually testing all of the functionalities of the system and making sure that no errors occurred during use.

The artefact was also synchronised with the company's webstore, which uses WooCommerce. It is synchronised using WooCommerce webhooks. To validate that

the artefact is properly synchronised with the webstore, the webhook functionality was tested. To test the webhooks and the e-stores synchronisation with the database, test orders were made on the WooCommerce staging site, each containing 1-5 different stamps from the database. The orders were then manually marked as 'completed' to trigger the webhooks.

3.2.4 Testing

To ensure the developed web application met both the functional and non-functional requirements, a set of Key Performance Indicators (KPI's) was defined to reflect core qualities of a secure and efficient system. These KPI's guided the testing process and were selected based on common practices and the system's real-world goals. Each KPI is something that could be measured directly.

The table below lists the KPI's used during the testing phase.

KPI	**Why It Matters**	**Benchmark / Standard**
CRUD Functionality	Ensures that basic database operations perform correctly	Create, Read, Update, and Delete operations must succeed
Input Validation Accuracy	Prevents incorrect or incomplete data from being saved	Should block invalid or missing input
Authentication Reliability	Ensures only authorized users can access the system	Must verify credentials, lock out brute-force attempts
Excel Export Accuracy	Allows the company to back up or reuse data in the same format as before	Should match old Excel layout
Webhook Synchronization	Keeps inventory up to date with the e-store in real time	Should update the correct stamp in database without issues
CRUD Operation Speed	Affects how responsive and usable the system feels when managing data	$<$ 0.5s for 30k, $<$ 2s for 100k entries (Defined in Section 1.3)
SQL Injection Resilience	Protects the database from malicious user input	Should reject unsafe inputs

Table 3.1: Defined KPI's used during testing

Each of these KPI's was tested through practical use cases and scenarios that reflect how the system would be used in a real environment. The KPI's were not derived from an existing testing framework but were instead defined on the system's functionality and the aspects that were considered most critical to evaluate. The following sections describe how we tested each feature in detail, including how we validated the system's performance, security, and reliability based on the defined KPI's.

To test the functionality of the web application, such as the different CRUD operations, Create, Read, Update, and Delete operations were conducted on test stamps. The Create operation was validated by creating new stamps and checking the database to ensure the stamp was correctly inserted. The Read operations were tested by simply displaying stamps in our stamp list and ensuring that the stamps were correctly displayed. The Update operation was validated by updating five different stamps and checking the database to ensure the stamps were correctly updated, and lastly, the Delete operation was validated by deleting five stamps and checking the database to ensure the stamps were deleted successfully.

Input validation was tested by creating stamps with missing required fields or incorrect formats, such as negative prices or quantities. Validation was used to make sure the data stayed consistent and logical, and to avoid errors caused by missing or invalid inputs such as negative prices. These checks were done both on the frontend and backend to ensure the data going into the system was correct.

Authentication was tested by attempting to login with both valid and invalid credentials to verify that the system responded appropriately in each case. Tests were also conducted to observe the behaviour after repeated failed login attempts. These tests were designed to confirm that the authentication system behaves securely and correctly under both normal and malicious scenarios. Reliable authentication is a critical component of web application security, and the test cases were built on OWASP authentication guidelines [14], which emphasise credential verification and resistance to brute-force attacks as standard expectations.

The *Generera Excel-Blad* page was tested and validated by generating Excel sheets before and after data changes, then verifying that the output format matched the company's previous system and included all expected fields.

Real-time synchronization between the WooCommerce e-store and the backend was tested by creating test orders using the WooCommerce staging website. Test orders were created and marked as 'complete' to simulate real transactions. Upon receiving the webhook notification, the backend was monitored to confirm that it processed the webhook notification, updated the database, and handled the data as expected. Webhook handling is critical to ensure inventory accuracy, especially when multiple customers interact with the store concurrently. Webhooks were chosen over polling due to their lower latency and real-time capabilities, which made them the preferred mechanism for the constructed system.

To test the non-functional aspects of the system such as security and performance, different tests were conducted. The security of the system was thoroughly

tested to ensure reliability and security throughout the system. SQL injection tests were performed by attempting to insert malicious JavaScript and SQL queries into the input fields. SQLMap was the tool used to simulate real-world injection attempts. SQL injections are one of the top web application security risks listed by OWASP in an article [15], and defending against it is considered a standard for any database-connected application.

Performance tests were conducted by populating the database with up to 100,000 stamps. Various operations, such as querying, filtering, inserting, and deleting records, were then performed using the frontend to evaluate whether the system remained responsive under high data loads. The read and filtering operations were also selected to reflect realistic relational query operations, such as multi-table joins and subqueries, which are common in data-driven systems. To reflect actual user interaction, total latency, including backend execution and frontend rendering, was measured for each operation. Benchmarks for acceptable performance were based on research mentioned in the **Purpose and Goals** section (see section 1.3), which defined expected CRUD operation times for databases of different sizes. Although the referenced study focused on backend performance, we applied its thresholds as general targets for user-perceived responsiveness in a full-stack web application. This reflects the importance of responsiveness in preventing delays during inventory tasks and ensuring the system remains efficient for day-to-day business operations. CRUD operations were considered successful if the total time remained within these defined limits.

Beyond basic functional and security tests, additional structured testing was conducted to assess the behaviour of the system under edge cases and less common usage scenarios. Each major feature was tested with both valid and invalid inputs through the user interface, and the outcomes were verified directly via database inspection and GUI feedback.

Read functionality was tested by applying various sorting and filtering criteria in the frontend and confirming that the displayed results matched expectations. The pagination functionality was tested to validate functionality and frontend rendering accuracy.

Delete operations included testing the deletion of stamps linked to multiple categories and recently added entries. The goal was to verify that data deletions were accurate and did not affect unrelated data.

Excel export functionality was tested by generating sheets before and after data changes, then verifying that the output format matched the company's previous system and Excel file structure.

3.2.5 Development environment and Collaboration tools

During the implementation of this project, various backend and frontend tools were utilized. Visual Studio Code *VS Code* served as the primary development environment due to its versatility, extensive extension support, and built-in tools for debugging.

To allow for easy code sharing and documentation, GitHub was used. GitHub is a proprietary developer platform that lets users store, manage, and share code in a centralised repository. GitHub uses branching to manage different features and tasks independently, ensuring that work can be done in parallel without conflict. Each user worked on separate branches, which were then reviewed and merged together.

3.2.6 Deployment

This section summarises the steps that are required to be taken to deploy the artefact.

The deployment of the database itself requires a standard MySQL database instance to be set up on Google Cloud SQL. The database instance requires public IP assignment to be enabled in order to allow external services to access the database.

Both the front-end and back-end parts of the application are run on two individual Docker containers. Dockerfile's for these containers can be seen in listings A.5 and A.7. When creating an instance on Cloud Run for each of these containers, the exposed port for each container specified in the Dockerfile has to be the same as the instance's exposed port.

The backend requires a requirements.txt file to list all of the Python packages that are required to be installed, which can be seen in listing A.6. The backend establishes an encrypted connection with the Cloud SQL database using the Cloud SQL Auth Proxy [16]. This is configured in the setup of the back-end instance on Cloud Run and automatically places a socket in the */cloudsql* directory when the container is deployed on Cloud Run, which allows the use of the proxy. The program then uses the socket together with the credentials of an authorised user to connect to the database. It is vital that both the back-end container and the database instance are placed under the same Google Cloud project, as this will allow the back-end to be connected to the database using a Google Cloud Connection.

Chapter 4

Results

The project resulted in a fully functional web application that fulfilled both the functional and most of the non-functional requirements. The GUI enabled stamp management with the use of drop-down inputs to ensure integrity (see figure A.3), while user authentication ensured only authorized personnel could access the system (see figure A.1). The stamp list implemented enabled users to view, update, and delete already existing stamps in the database (see figure A.2). Real-time WooCommerce synchronization ensured inventory and sales data remained accurate across platforms. Performance testing confirmed the system's ability to handle large-scale operations without degradation of performance. Security tests confirmed the protection of the system against SQL injections.

The following results are presented according to the KPI's defined in Section 3.2.4, ensuring that each key aspect of the system was validated.

4.1 Functional Test Results

Functional tests were performed to validate the functionality of the constructed web application and the different pages included in the web application (see section 1.4).

4.1.1 CRUD Operations

All CRUD operations were thoroughly tested through the system's user interface to verify that the application met its functional requirements (see section 1.4). These tests were performed across different parts of the application, such as the different pages constructed to allow for these CRUD operations.

Create operations were tested by inserting new stamps using various combinations of input values. These tests included valid data, missing required fields, and invalid numeric values like negative prices. The system correctly handled these cases by accepting valid input and rejecting invalid entries through frontend and backend validation. Cases where important values were missing were successfully rejected. The creation of new entries in the different attributes was also tested and

validated. Whenever new entries were added, they were correctly displayed in the drop-downs for that attribute. Whenever the same stamp is added multiple times, separate entries are created in the database, mirroring the functionality the company used in their Excel file.

Read operations were verified through the stamp list interface. The application successfully received and displayed the stamp records from the database. Pagination and sorting were also confirmed to be functional and responsive, ensuring users could navigate and view stamps efficiently.

Update operations were tested by selecting individual stamps from the list and modifying their attributes. In each case, changes were submitted through the interface and confirmed to be correctly saved in the database. The relational integrity of updated entries, including category and attribute references, was maintained after each operation.

Delete operations were performed by removing stamps from the stamp list page. Tests included deleting stamps with different attributes, both new and old ones. Each deletion was successfully executed, and all associated relationships in junction tables were also removed, confirming no orphaned data remained in the database.

Input validations were verified at both the frontend and backend levels, ensuring invalid data, such as negative prices or amounts, were rejected before reaching the database.

4.1.2 Authentication

Authentication tests were conducted to validate the reliability and security of the login system. Both valid and invalid credentials were tested to ensure the system correctly handles authentication requests. Successful login attempts were confirmed by verifying that users were redirected to the main menu and that their session was stored on the backend, allowing seamless access upon page refresh. Attempts with incorrect usernames or passwords resulted in appropriate error messages, and access to protected routes was denied. Whenever a user inserts the wrong username and/or password, the same error message is shown, ensuring that no information about how far into the login process they've gotten is shown.

4.1.3 Excel Generation

The Excel Sheet Generation page successfully produced accurate and complex exports of the database. The exported files consistently reflected all current data in the system, including after updates and deletions. The format matched the company's previous Excel structure, and all expected fields were included without data loss or formatting issues.

4.1.4 Webhook Synchronization

Real-time synchronisation with WooCommerce was verified by ensuring that whenever a webhook was triggered, the *StoreStamps* and *SoldStamps* tables were correctly updated. These tests confirmed that the system maintained consistency and real-time synchronisation with the e-store. In all five test cases, stock quantities in the *Store_Stamps* table were appropriately reduced, and corresponding sales records were inserted into the *Sold_Stamps* table, indicating successful synchronization.

4.2 Non-Functional Test Results

4.2.1 Data Integrity and Consistency

To validate the reliability of the system, several tests were conducted to ensure data integrity and consistency across all operations. Foreign key constraints were tested to confirm that relationships between tables, such as stamps and their associated attributes, enforced referential integrity. During the performance testing for stamp insertion, each operation was validated to ensure correct relational integrity across all connected tables. For example, when a new stamp was created, the stamp was verified to have its assigned categories, country, size, etc, exist in their respective tables. This ensured that all relationships adhered to the database schema (see 3.2) and verified that the system's integrity and consistency stayed persistent.

4.2.2 Security

To test the system's resilience against malicious attacks such as SQL injections, tests have been performed on the system to determine whether the system has any potential security vulnerabilities.

For the SQL injection tests, SQLmap was used to test the application for vulnerabilities. Given the fact that the majority of the endpoints take numbers as arguments, and make sure that this is what is given as an argument, the majority of the endpoints won't be susceptible to SQL injections. But, out of the few endpoints that take plain text as arguments, the SQLmap tool was used. SQLMap performs commands such as "1; DROP TABLE users–" to try and manipulate the server and "1' AND (SELECT 1 FROM dual WHERE database() LIKE '%')– -" to try and get information about the database structure out of the system. The tool could not find any parts of the application that were susceptible to SQL injections.

4.2.3 CRUD Operation

Simple CRUD operations, such as create, read, and delete, were all performed to measure the performance of the system. To measure the time it took for the frontend to render all the stamps, we used React's built-in performance tools to programmatically track the components' rendering time. For the backend, we used

time.perf_counter() to record timestamps at the start and end of the stamp-fetching operation, then by subtracting the differences, a measurement can be found. The average total time is the time it took to fetch multiple stamps and render them on the frontend. The tables below show the average time it took to fetch between 100 and 2000 stamps. Two different tests were conducted: one where the database was populated with up to 30,000 stamps, the number currently stored in the company's Excel file, and another with up to 100,000 stamps to evaluate the scalability of the database. To populate the database with 100,000 different stamps, a simple script was written that constructed stamps with attributes that were chosen at random and then inserted into the database.

Table 4.1: Read operation measurement with ~30,000 stamps

Amount	Avg. Total Time *(ms)*	Avg. Backend Time *(ms)*	Avg. Frontend Time *(ms)*
100	49.54	37.68	11.86
500	82.28	39.32	42.96
1000	120.17	51.95	68.22
2000	192.85	60.89	131.96

Table 4.2: Read operation measurement with ~100,000 stamps

Amount	Avg. Total Time *(ms)*	Avg. Backend Time *(ms)*	Avg. Frontend Time *(ms)*
100	50.17	39.56	10.61
500	83.87	44.42	39.45
1000	133.82	50.26	83.56
2000	220.87	82.68	138.19

The filtering feature was also tested and measured to validate the system's ability to perform subqueries. Below is a table showing the time measured when fetching all of the stamps with *Lichtenstein* as their country attribute, both for a database with 30,000 and 100,000.

Table 4.3: Filtering functionality (subqueries)

Database Size	Amount	Avg. Backend Time *(ms)*	Avg. Frontend Time *(ms)*
30,000	124	50.04	25.48
100,000	153	58.86	37.36

Each of the create, delete and update operations was done five times. Below are two tables showing the average time it took for each component to perform the different operations on a database with 30,000 stamps and, respectively, 100,000 stamps. The average frontend time is the time it took from the moment that the frontend sent a request to the backend to the moment that it received and handled the response from the backend. The average backend time is the time it took for the backend to process the received data, perform the operation, and return a response to the frontend.

Table 4.4: Create, Delete and Update operation measurements with ~30,000 stamps

Operation	Avg. Backend Time *(ms)*	Avg. Frontend Time *(ms)*
Create	452.3	760.2
Delete	155.8	411.4
Update	318.3	961.2

Table 4.5: Create, Delete and Update operation measurements with ~100,000 stamps

Operation	Avg. Backend Time *(ms)*	Avg. Frontend Time *(ms)*
Create	486.2	842.9
Delete	158.7	413.8
Update	321.4	1109.4

Chapter 5

Discussion

Overall, the system fulfilled most of the KPIs defined in Section 3.2.4, meeting both functional and non-functional requirements. Functional KPIs, such as CRUD accuracy, input validation, authentication, and real-time synchronization, were successfully validated through system tests. Non-functional KPIs, such as security aspects like SQL injection resilience, were satisfied based on relevant benchmarks. However, some performance-related KPIs, especially some CRUD operations under higher data volumes, were not fully achieved and are discussed further below.

5.1 Analysis of functional results

5.1.1 CRUD Operations

All CRUD operations were successfully validated through the system's graphical interface and database and met all functional requirements defined in the **Requirements** section (see Section 1.4).

Create operations correctly handled both valid and invalid input through frontend and backend validation, preventing malformed data from reaching the database. New attribute entries appeared immediately in drop-downs, ensuring a responsive and dynamic interface.

Read operations were verified through the paginated stamp list interface, which allowed for sorting and filtering by various attributes. The results consistently matched the database contents, confirming correct retrieval logic and frontend rendering. Pagination further ensured that stamp display remained responsive even with large datasets.

For update operations, the system correctly handled submitted changes to the backend, and all modifications were saved in the database. The application maintained referential integrity, for example, updates to a stamp's categories or countries reflected accurately across related tables without orphaned or duplicate entries.

Delete operations were tested on stamps of varying complexity. Deletion consistently removed the record and associated entries in many-to-many linked tables.

This confirmed the systems ability to maintain relational integrity and avoid orphaned entries after deletions.

5.1.2 Authentication

The authentication system met all functional requirements, with successful logins creating persistent sessions and failed attempts returning consistent error messages without revealing system details.

5.1.3 Webhook Synchronization

The webhook tests demonstrated consistent updates to inventory and sales records upon receiving real-time order events. This behaviour aligns with the systems functional requirement of supporting real-time data management. Reliable webhook handling is especially important in retail systems, where poor inventory management can lead to overselling.

5.2 Analysis of non-functional results

5.2.1 CRUD Operations performance

These results were evaluated against the benchmark criteria defined in the **Purpose and Goals** (see Section 1.3) and outlined in the **Method Testing** section (see Section 3.2.4. CRUD operations were considered successful if the total time, including backend execution and frontend rendering, remained within 0.5 seconds for 30,000 records and within 2 seconds for 100,000 records.

Our system efficiently fetched and displayed up to 2,000 stamps in approximately 193ms with a database containing 30,000 stamps and 220ms with 100,000 stamps (see tables 4.1 and 4.3).

When compared to academic benchmarks for single-table read operations, typically achieving 300-500 ms for up to 50,000 records [2], despite involving multi-table joins between stamps, countries, categories, etc. These additional relationships and join operations naturally increase the query complexity and execution time, yet our implementation remains within both the non-functional requirements and external standards. Similarly, the filtering functionality was tested under realistic conditions and performed efficiently, with average backend execution times of around 50ms for a database with 30,000 stamps and 59ms for 100,000 stamps. Although the queries are not written as explicit SQL subqueries, the underlying logic achieves the same result. Since all tests passed without performance issues, it can be concluded that the system is effective at handling complex queries, including multi-table joins and subqueries.

Create, update, and delete operations all met performance goals at the backend level. However, for update and create operations, the total frontend and backend time exceeded the threshold. Since frontend performance depends on the user's

local device and browser environment, and delete and update operations require more complex queries, this increase in time was expected.

This means that while the backend fulfilled the defined performance KPI for CRUD operations, the total end-to-end operation time, especially for create and update actions, only partially satisfied the overall KPI when considering frontend delays. This highlights a limitation in meeting the non-functional performance benchmark defined in Section 3.2.4.

Beyond measuring performance against set KPI's, it was also important to examine how the system handled increasing data volumes, particularly whether query times scaled linearly as database size increased. Although a linear relationship between database size and query time might be expected, this was not the case in the constructed system. As demonstrated in Tables 4.2.3, the time difference between querying a database with 30,000 entries versus 100,000 entries was minimal. This can be attributed to the system's use of pagination, which limits the number of rows fetched per request (e.g., 1000). As a result, the database engine avoids full table scans, ensuring consistent performance regardless of total data size. Furthermore, the create, update, and delete operations affect only single entries and benefit from transactional isolation, reducing performance impact even at higher data volumes.

5.2.2 Security

The system's resilience against malicious attacks, such as SQL injections, was successful. By running SQLMap, we confirmed all endpoints were resilient to SQL injection attacks, validating the effectiveness of SQLAlchemy's parameterised queries.

While the system successfully defends against SQL injection attacks, other common web vulnerabilities such as Cross-Site Scripting *(XSS)* and Cross-Site Request Forgery *(CSRF)* were not addressed in this project due to time constraints. However, React already escapes values in JSX by default, which helps prevent XSS vulnerabilities [17]. Flask also supports CSRF protection through extensions such as Flask-WTF [18]. These protections were not implemented in the current version but should be considered in future development to ensure a more comprehensive, secure system.

5.2.3 Database Design and Normalisation

The database was designed to satisfy the requirements for 3NF (see 2.2), optimizing both data integrity and performance. The database was designed to enforce single and atomic values in each cell (1NF), required all non-key attributes to depend fully on primary keys (2NF), and removed transitive dependencies (3NF). Strict foreign key constraints were enforced across all entity relationships, ensuring referential integrity throughout the system. Additionally, a recursive hierarchical category relationship was implemented by using *Parent_ID* references. This approach was chosen to address the company's complex stamp category system, where stamps

could belong to multiple nested categories (e.g., Europe, Sweden, Halland). This approach reduced schema complexity by eliminating the need for multiple category tables and minimising data redundancy when stamps shared common parent categories.

5.3 Development Process Reflection

The development process was generally effective, even though it was not strictly followed as originally planned. While we officially envisioned developing each component, such as the database, frontend, and backend, one after the other, the actual workflow turned out to be more iterative and overlapping. The workflow went from working with the frontend, styling it and making sure it is easy to use, to implementing a new function in the backend the next hour. This workflow worked better as both the frontend and backend part had to be updated when implementing a new function in order to quickly test it and fix potential bugs.

5.4 Societal Aspects

Our stamp management system brings several important social and ethical benefits. First and foremost, by replacing error-prone Excel spreadsheets with a secure digital system, we're helping prevent business mistakes that could lead to financial losses or customer dissatisfaction. The system's security features protect both the company's data and potentially sensitive customer information from breaches, an essential ethical requirement in today's digital world.

While developing for a specific small business, our solution demonstrates how digital transformation can help similar small enterprises become more sustainable and competitive. The time employees save from manual data entry can be redirected towards better customer service or business growth. The web-based design also supports remote work flexibility, contributing to better work-life balance.

The developed system shows how even targeted technological solutions can have a wider positive effect. By making business operations more efficient and secure, we are contributing to a broader trend of helping small businesses thrive in the digital economy while maintaining ethical standards for data protection and workplace flexibility.

Chapter 6

Conclusion

The problems that have been solved in this work originate from the company's prior dependence on manually editing Excel files for managing their stamps. It led to big-time consumption trying to find relevant data, inaccurate data being input, and came with the inability to sync the data storage with other parts of the operation, such as the company's web store. The company also found that it was difficult to manage the growing size of the file they used.

This resulted in a web-based data storage solution. This consisted of a system where stamps and related data are stored in a relational database hosted on Google Cloud SQL and managed with MySQL DBMS. Data is accessed and manipulated through a GUI that is hosted on the web and was developed using ReactJS. A back end web based system has been developed using Flask and SQLAlchemy to bridge the gap between the GUI and the database and support extra functionality, like the connection with the web store using webhooks.

Relevant data is now easier to find by being able to sort and filter the stamps based on different data fields. The problem with data inaccuracies stemming from the user error that comes with manual data input is now solved with the help of predefined input options in the form of drop-downs. The integration with the webstore now also ensure that the availability state of the state is properly reflected in the database.

6.1 Future Work

One thing that could be added to the artefact to improve it could be adding support for bidding for stamps on Tradera and having the data management system update the price of the stamp in real time in the database. This would make it possible to sync up both Tradera and the company's own webstore, so that they can host the same product on both platforms.

XSS vulnerabilities could also be addressed in the future, with tests to confirm the system's resilience against attacks other than SQL injections.

Bibliography

[1] P. Education, *Introduction to databases*, Lecture slide, Database course, Halmstad University, 2014.

[2] A. M. Bonteanu and C. Tudose, "Performance analysis and improvement for crud operations in relational databases from java programs using jpa, hibernate, spring data jpa," *Applied Sciences*, vol. 14, no. 7, 2024, ISSN: 2076-3417. DOI: 10.3390/app14072743. [Online]. Available: https://www.mdpi.com/2076-3417/14/7/2743.

[3] J. Poudel, "Library management system with react.js," Bachelor's thesis, Centria University of Applied Sciences, 2023-06-14T09:16:56Z. [Online]. Available: https://www.theseus.fi/handle/10024/804486.

[4] M. R. Mufid, A. Basofi, M. U. H. Al Rasyid, I. F. Rochimansyah, and A. rokhim, "Design an mvc model using python for flask framework development," in *2019 International Electronics Symposium (IES)*, 2019, pp. 214–219. DOI: 10.1109/ELECSYM.2019.8901656.

[5] K. Velavan, *React vs flask*, Accessed: 2025-02-26, 2023. [Online]. Available: https://medium.com/@gkavinvelavan/react-vs-flask-957c2799b0f7.

[6] E. Nilsson and D. Demir, "Performance comparison of rest vs graphql in different web environments: Node.js and python," Ph.D. dissertation, 2023. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:lnu:diva-121877.

[7] Jun. 2025. [Online]. Available: https://aws.amazon.com/what-is/web-application/.

[8] 2025. [Online]. Available: https://foldoc.org/three-tier.

[9] N. Idris, C. F. Mohd Foozy, and P. Shamala, "A generic review of web technology: Django and flask," *International Journal of Advanced Science Computing and Engineering*, vol. 2, pp. 34–40, May 2021. DOI: 10.30630/ijasce.2.3.29.

[10] Flask Team, *Flask-login*, Accessed: 2025-03-04. [Online]. Available: https://flask-login.readthedocs.io/en/latest/.

[11] P. Sultana, N. Sharma, N. Nagendhiran, G. Pathak, and A. Pandey, "Prevention of sql injection using a comprehensive input sanitization methodology," in Jan. 2023, ISBN: 9781643683607. DOI: 10.3233/ATDE221269.

[12] Flask Team, *Flask-sqlalchemy*, Accessed: 2025-03-04, 2010. [Online]. Available: https://flask-sqlalchemy.readthedocs.io/en/stable/.

[13] sqlmap, *Sqlmap - automatic sql injection and database takeover tool*, Accessed: 2025-05-12. [Online]. Available: https://sqlmap.org/.

[14] *Owasp authentication*, 2025. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html.

[15] *Owasp top ten*, 2025. [Online]. Available: https://owasp.org/www-project-top-ten/.

[16] 2025. [Online]. Available: https://cloud.google.com/sql/docs/mysql/sql-proxy.

[17] *Reactjs xss protection*, 2021. [Online]. Available: https://legacy.reactjs.org/docs/dom-elements.html#dangerouslysetinnerhtml.

[18] *Flask-wtf-documentation*, 2025. [Online]. Available: https://flask-wtf.readthedocs.io/en/1.2.x/.

Appendix A

GUI Images

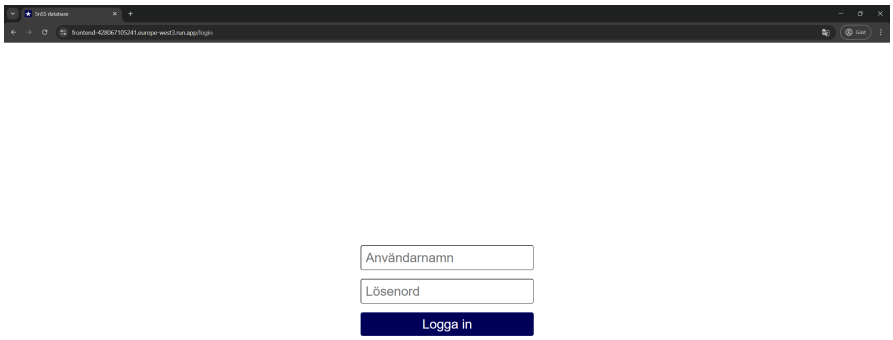

Figure A.1: The login page users are directed to when entering the web application.

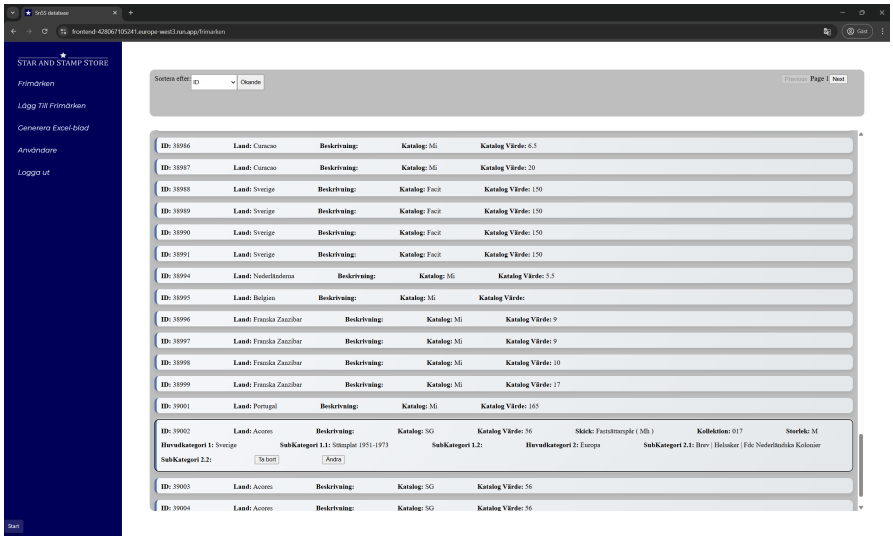

Figure A.2: The stamp list page where users can alter, delete and view stamps.

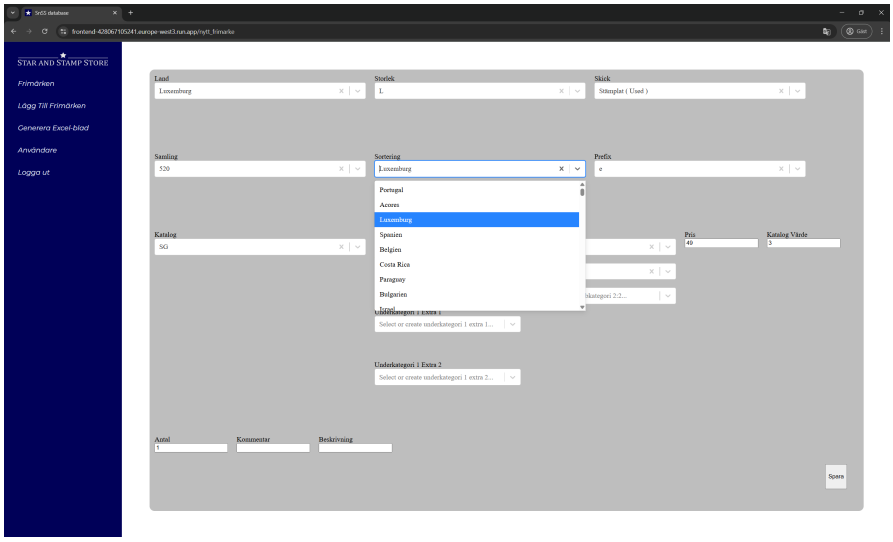

Figure A.3: The page where users can insert new stamps, with an example of how the drop-down inputs look.

Listing A.1: An example API endpoint that handles HTTP requests

```
from flask import request, jsonify, Response

@app.route("/stamps", methods=["GET","POST"])
@login_required
def stamps() -> Response:
    if request.method == "GET":
        try:
            #Fetch stamp info
            return jsonify("stamp_info")
        except Exception as e:
            return jsonify({"error": str(e)}), 500
    if request.method == "POST":
        try:
            #Insert new stamp
            return jsonify("Successful message")
        except Exception as e:
            return jsonify({"error": str(e)}), 400
```

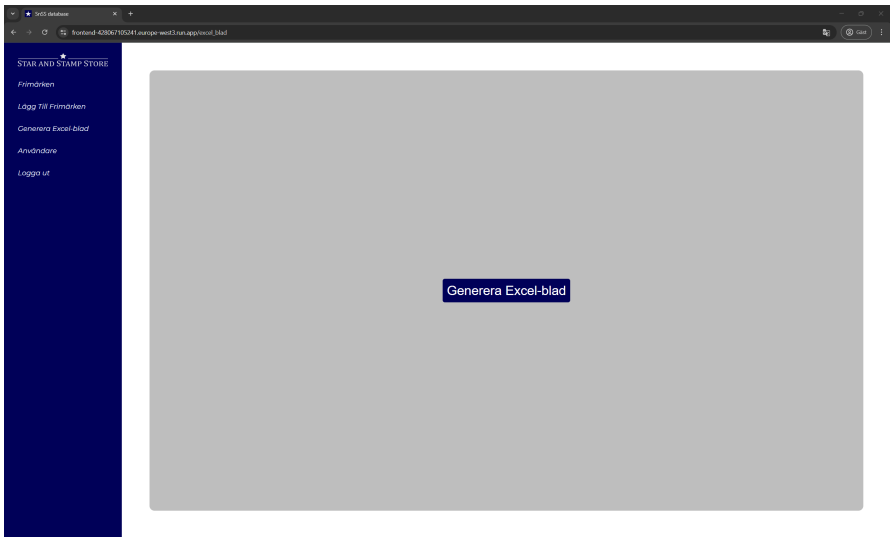

Figure A.4: The page where users can generate Excel sheets with information about the stamps from the database.

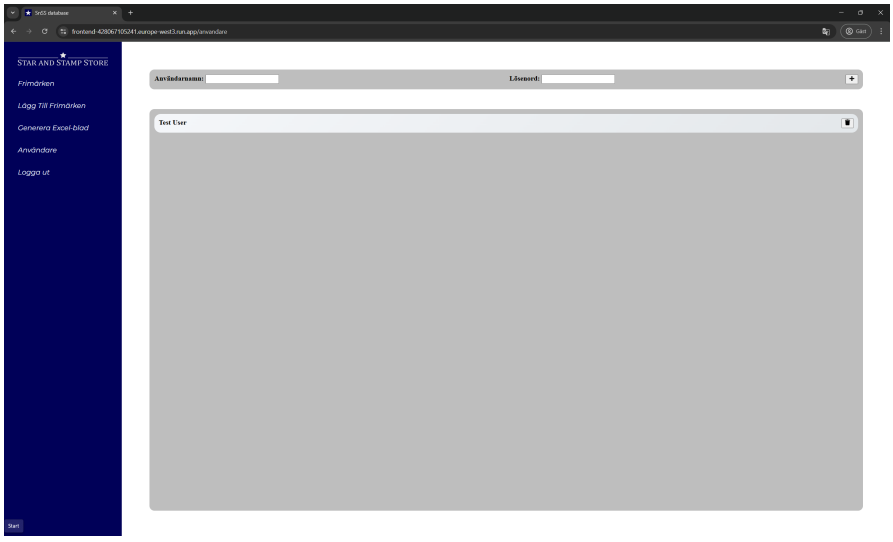

Figure A.5: The page where the admin can manage users in the system.

Listing A.2: Data migration Excel Sheet column handler

```
from pandas import DataFrame, read_excel


class ExcelSheet:
    def __init__(self, work_sheet: DataFrame):
        self.work_sheet = work_sheet

    def get(self, col: str, row: int):
        col_index = ExcelSheet.convert_letters_to_index(col)
        cell = self.work_sheet.iloc[row, col_index]
        if cell.value is None: return None
        if type(cell.value) == str: return cell.value.lower()
        return cell.value

    @staticmethod
    def convert_letters_to_index(letters: str) -> int:
        return sum(
            26 ** (len(letters) - index - 1) * (ord(character) - 64)
            for index, character in enumerate(letters.upper())
        ) - 1

df = read_excel("filename.xlsx", sheet_name="Sheet Name")
sheet = ExcelSheet(df)

def insert_simple_collumn(col: str, table) -> dict[str, int]:
    entries = {}
    try:
        for row in range(1, AMOUNT + 1):
            value = sheet.get(col, row)
            if not value or value in entries.keys(): continue
            entry = table.make_entry(value.capitalize())
            if not entry: continue
            database.session.add(entry)
            entries[value] = entry
    except Exception as e:
        database.session.rollback()
        raise e
    return entries
```

Listing A.3: An example of how to authenticate webhooks

```
import hmac
import hashlib
import base64

def authenticate_request(secret: bytes) -> bool:
    signature = request.headers.get('X-WC-Webhook-Signature')
    data = request.get_data()
    computed_signature = base64.b64encode(
    hmac.new(secret, data, hashlib.sha256).digest()).decode()
    return hmac.compare_digest(signature, computed_signature)
```

Listing A.4: An example of a webhook endpoint

```
@app.route('/webhook/order_updated', methods=['POST'])
def order_updated() -> Response:
    try:
        secret = os.getenv('WOOCOMMERCE_ORDER_UPDATED_SECRET').encode()
        if not authenticate_request(secret):
            return jsonify({"error": "Unauthorized"}), 401
        data = request.get_json()
        if data['status'] != 'completed':
            return jsonify({"error": "Order not completed"}), 400
        date = datetime.strptime(data['date_completed'], "%Y-%m-%dT%H:%M:%S")[
            handle_stamp_sold(
                stamp_id = int(item['sku'].split('_')[-1]),
                amount = int(item['quantity']),
                sold_date = date
            ) for item in data['line_items']]
    except Exception as e:
        database.session.rollback()
        return jsonify({"error": str(e)}), 500
    return jsonify({"message": "Order processed successfully"}), 200
```

Listing A.5: Dockerfile for building and running the backend

```
# A base image that includes Python
FROM python:3.12-slim

# Sets up a working directory in the container for the program
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

# Sets environment variables
ENV FLASK_APP=main.py
ENV FLASK_ENV=production

# Copies the code into the container
COPY . .

# Creates a directory for the Cloud SQL proxy
RUN mkdir /cloudsql

# Installs any Python dependencies listed in 'requirements.txt'
RUN pip install -r requirements.txt

# Exposes the port the program runs on
EXPOSE 8080

# Sets the command to run the program
CMD ["python", "-u", "main.py"]
```

Listing A.6: The requirements.txt file for the backend application

```
Flask==3.1.0
flask_cors==5.0.1
flask_sqlalchemy==3.1.1
Flask-Login==0.6.3
PyMySQL==1.1.1
python-dotenv==1.0.1
SQLAlchemy==2.0.38
bcrypt==4.3.0
openpyxl==3.1.5
pandas==2.2.3
```

Listing A.7: Dockerfile for building and running the frontend

```
# A base image that includes Node.js
FROM node:20-alpine

# Sets up a working directory in the container for the programr
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

# Copies the code into the container
COPY . .

# Set the PYTHONPATH environment variable, which is occasionally necessary for
# certain node packages
ENV PYTHONPATH=${PYTHONPATH}:${PWD}

# Sets the environment variable for the application's port
ENV PORT 8080

# Installs 'serve'
RUN npm install -g serve

# Installs all the other node modules required by the React app
RUN npm install

# Builds the React app
RUN npm run build

# Sets the command to run the program
CMD ["serve", "-s", "-l", "8080", "./dist"]
```