Inventory Management System

Making Renting and Charching effortless

Felix Huther, Jingya Zhao, Kandaker Majharul Islam, Philipp Becker, Sven Lepper

A collaborative project involving
Hochschule Darmstadt (Germany)
& Centria University of Applied Sciences (Finnland)

Spring/Summer Term 2024

1 Give the project a name / Cover page: (Ifty)

1. Name of Project
2. This is a project between: 2 partners: Centria, h_da
3. date (spring/summer term 24)
4. names of team members

Contents

2 Quote example

According to **(author?)** [1], ...

3 Content page Ifdi

4 Introduction Philipp

1. goal / problem that should be solved
2. how did we divide up the members of the class into teams; introduce the teams (tasks, people, tasks of individuals)- keep this short

5 Research / State of the Art / alternatively: Motivation (Ting)

(keep this short and possibly replace it by a motivation why you chose this topic)

6 Methodology (Felix)

1. which are the steps that are taken in which order to reach the result
2. which steps are taken
3. which tools were used (keep this short)
4. always give reasons for WHY you chose certain steps
5. always give reasons for WHY you chose certain tools

7 Software Design

Written by: Sven

7.1 Introduction

This chapter delves into the intricate details of the software design employed in the development of the locker system for university environments. A robust and efficient software architecture is pivotal in ensuring the seamless functioning and user satisfaction of the system. In this section, we provide an overview of the architectural approach, technologies, and frameworks chosen for the implementation of the locker system software, highlighting their significance and relevance to the project objectives.

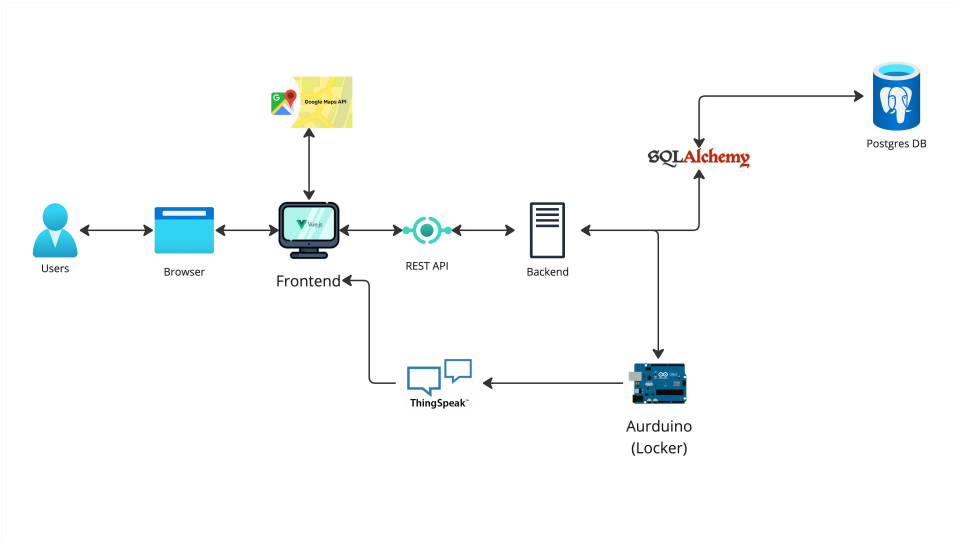

Figure 1: Software Design Diagram

The image above depicts a high-level overview of the software design architecture, illustrating the interaction between various components and their roles within the system. Throughout this chapter, we elucidate the rationale behind each architectural decision and technology selection, elucidating

their benefits and contributions to the overall functionality and performance of the locker system.

Now, let's delve into the intricacies of the software design, beginning with an exploration of the architectural approach adopted for the project.

7.2 Architectural Approach

The adoption of a modern microservices architecture, with clear separation between frontend and backend components, offers several advantages for the locker system project. By decoupling these layers and enabling communication via a RESTful API, we achieve enhanced modularity, scalability, and flexibility. This modular approach facilitates independent development and deployment of frontend and backend services, enabling rapid iteration and evolution of the system. Moreover, the RESTful API design simplifies integration with third-party services and future expansion of functionality, ensuring adaptability to changing requirements and technological advancements.

7.3 Frontend Technologies

Vue.js was selected as the frontend framework due to its lightweight nature, simplicity, and extensive ecosystem of plugins and libraries. Its reactive data binding and component-based architecture enable the creation of dynamic and interactive user interfaces, enhancing the user experience. Additionally, the integration of Tailwind CSS and Bootstrap provides a comprehensive set of styling utilities and pre-designed components, enabling rapid prototyping and ensuring consistent design across different devices and screen sizes. The use of these frontend technologies not only accelerates development but also enhances maintainability and scalability, making them well-suited for a complex application like the locker system.

7.4 Google Maps Integration

The integration of the Google Maps API enriches the user experience by providing visual representation of locker locations. This feature enhances user convenience and navigation, particularly in large university campuses where locker locations may not be readily apparent. By leveraging the Google Maps API, users can easily locate their rented devices, thereby reducing frustration and improving overall satisfaction with the service.

7.5 Backend Technologies

FastAPI was chosen as the backend framework due to its high performance, asynchronous capabilities, and intuitive API design. Its built-in support for asynchronous programming enables efficient handling of concurrent requests, ensuring optimal responsiveness and scalability, especially under heavy loads. Additionally, FastAPI's automatic generation of OpenAPI documentation simplifies API documentation and client integration, enhancing developer productivity and collaboration. The use of FastAPI aligns with industry best practices and standards, ensuring the reliability, security, and maintainability of the locker system backend.

7.6 Database Management

The utilization of SQLAlchemy as the ORM tool offers numerous benefits for database management within the locker system project. By abstracting away the complexities of database interaction, SQLAlchemy enhances developer productivity and code maintainability, while also mitigating the risk of SQL injection vulnerabilities. Furthermore, Alembic provides seamless database schema versioning and migration capabilities, facilitating continuous evolution and refinement of the database schema over time. These features ensure data integrity, consistency, and scalability, making

SQLAlchemy and Alembic well-suited for managing the relational database backend of the locker system.

7.7 Database Choice

PostgreSQL was selected as the underlying database management system for its robustness, reliability, and advanced feature set. Its support for ACID transactions, data integrity constraints, and extensible data types ensures the consistency and integrity of stored data, critical for a mission-critical application like the locker system. Additionally, PostgreSQL's scalability and performance optimizations make it capable of handling large volumes of concurrent transactions and complex queries, making it an ideal choice for the storage and retrieval of locker rental and user data.

7.8 Locker Controls

The integration of Arduino-based locker controls provides a cost-effective and customizable solution for managing locker access and device rentals. Arduino's open-source hardware platform, coupled with its rich ecosystem of sensors and actuators, enables the implementation of tailored locker control mechanisms suited to the specific requirements of the locker system. By exposing REST APIs for communication with the backend, Arduino controllers facilitate real-time updates and authentication checks, ensuring secure and reliable access to rented devices.

7.9 Charging Data Management

ThingSpeak serves as an IoT analytics platform for collecting, storing, and visualizing charging data generated by the Arduino controllers. Its cloud-based infrastructure and user-friendly interface simplify data management and analysis, enabling administrators to monitor charging activities and

usage patterns in real-time. The integration of ThingSpeak with the frontend enables the seamless transmission of processed data for visualization, empowering administrators to make informed decisions regarding resource allocation and system optimization.

7.10 Hosting and Deployment

The successful deployment and hosting of the locker system components are crucial for ensuring accessibility, scalability, and reliability. In this section, we elucidate the deployment strategies employed for the frontend, backend, and database components, leveraging cloud-based solutions and containerization technologies for seamless deployment and management.

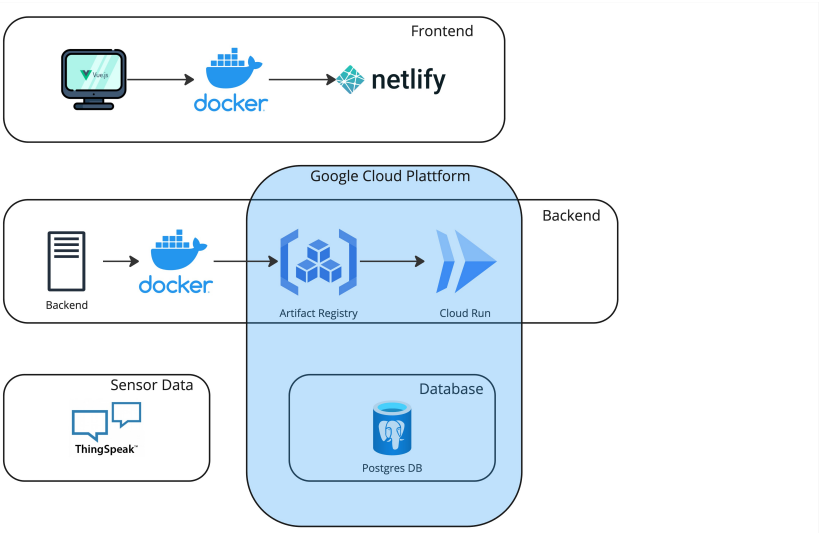

Figure 2: Overview of Deployment Processes

7.10.1 Frontend Deployment

The frontend of the locker system is packaged into a Docker container to encapsulate its dependencies and environment. This containerized approach

ensures consistency and portability across different environments, mitigating potential compatibility issues. Subsequently, the Docker container is deployed to Netlify, a popular platform for hosting static websites and web applications. Netlify provides a straightforward deployment process, seamless integration with Git repositories, and automatic build and deployment pipelines. Additionally, Netlify assigns a public address to the deployed frontend, enabling users to access the locker system via the internet with ease.

7.10.2 Backend Deployment

Similar to the frontend, the backend services are containerized using Docker to streamline deployment and management. Once the backend Docker image is built, it is pushed to Google Cloud's Artifact Registry, a managed service for storing container images securely. Artifact Registry ensures version control, access control, and image vulnerability scanning, enhancing the security and reliability of the deployment process. Subsequently, the containerized backend services are deployed to Google Cloud Run, a fully managed serverless platform for running containerized applications. Google Cloud Run automatically scales the backend services based on demand, ensuring optimal performance and cost-efficiency. Furthermore, Google Cloud Run assigns a public address to the deployed backend services, facilitating seamless communication with the frontend and other system components over the internet.

7.10.3 Database Hosting

The PostgreSQL database used by the locker system is hosted on Google Cloud SQL, a fully managed relational database service. Google Cloud SQL offers automatic backups, high availability, and scalability, relieving the

burden of database administration and maintenance. By leveraging Google Cloud SQL, we ensure data durability, reliability, and performance, critical for the storage and retrieval of locker rental and user data. Additionally, Google Cloud SQL provides seamless integration with other Google Cloud services, simplifying data management and access control.

7.10.4 Sensor Data Management

The sensor data collected by the Arduino controllers is transmitted to ThingSpeak, a cloud-based IoT analytics platform. ThingSpeak provides a scalable and reliable infrastructure for storing, analyzing, and visualizing sensor data in real-time. As a Software as a Service (SaaS) solution, ThingSpeak eliminates the need for hosting and infrastructure management, allowing developers to focus on application logic and data analysis. By leveraging ThingSpeak, we ensure efficient management and utilization of the sensor data.

7.11 Conclusion

In summary, the deployment and hosting of the locker system components leverage cloud-based solutions and containerization technologies to ensure accessibility, scalability, and reliability. By utilizing platforms such as Netlify, Google Cloud, and ThingSpeak, we streamline the deployment process, enhance security and performance, and enable seamless integration between different system components. This comprehensive approach to hosting and deployment underscores the importance of utilizing cloud-native technologies and services for building robust and scalable applications in modern computing environments.

The software design of the locker system incorporates a carefully selected set of technologies and architectural principles tailored to the specific

requirements and challenges of managing locker rentals and device access in university environments. Many of these technologies were introduced in lectures at our university, and while there may be other options available, we opted for those recommended by the university and with which we had prior experience. By leveraging modern frameworks, platforms, and best practices, the system delivers enhanced functionality, scalability, and user experience, ensuring its effectiveness and viability in real-world deployments.

8 Implementation

1. how did we reach the result / coding (snippets)
2. screen shots

8.1 Frontend

Written by: Philipp

8.2 Backend

Written by: Sven

8.3 Backend Implementation

This subsection provides an overview of the implementation details for the backend of the locker system. It covers the technologies, frameworks, and development environment used in the development process.

8.3.1 Project Setup

The backend part of the locker system project is developed using FastAPI and SQLAlchemy/alembic. These technologies were chosen for their robustness, scalability, and ease of use in building web APIs and managing database migrations.

The development environment setup involves several steps to ensure a smooth workflow. Firstly, it requires a local Postgres database instance to be running, with specific configuration settings provided via environment variables. These variables include the database username, password, host, port, and name.

The project setup also involves cloning the backend repository and configuring the environment variables as necessary. A virtual environment is

then created to isolate dependencies, followed by the installation of required packages specified in the ‘requirements.txt‘ file using pip.

Once the development environment is set up, the application can be run locally by activating the virtual environment and starting the application. Additionally, database migrations can be managed using alembic, with commands provided to generate and apply migration scripts.

8.3.2 List of Important Libraries and Versions

The following list includes the most important libraries used in the backend implementation along with their versions:

- **FastAPI** (`fastapi==0.110.2`): FastAPI is a modern, fast (high-performance) web framework for building APIs with Python 3.7+ based on standard Python type hints.
- **SQLAlchemy** (`SQLAlchemy==2.0.29`): SQLAlchemy is the Python SQL toolkit and Object-Relational Mapper that gives application developers the full power and flexibility of SQL.
- **Alembic** (`alembic==1.13.1`): Alembic is a lightweight database migration tool for usage with the SQLAlchemy Database Toolkit for Python.
- **Pydantic** (`pydantic==2.7.1`): Pydantic is a data validation and settings management using Python type annotations.
- **uvicorn** (`uvicorn==0.29.0`): Uvicorn is a lightning-fast ASGI server implementation, using uvloop and httptools.
- **psycopg2-binary**: Psycopg is the most popular PostgreSQL database adapter for the Python programming language.

8.3.3 Request Handling in FastAPI and SQLAlchemy

A crucial aspect of understanding the backend implementation is to grasp how FastAPI and SQLAlchemy handle incoming requests and interact with the database. The following diagram illustrates the request handling process:

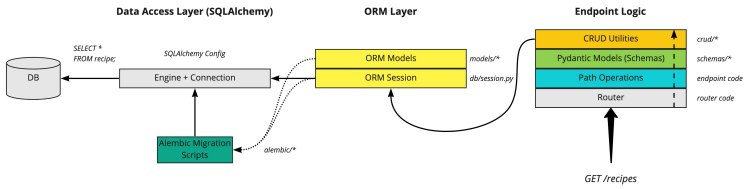

Figure 3: Request Handling in FastAPI and SQLAlchemy **(author?)** [2]

This diagram visually depicts the sequence of steps involved in processing a client request, starting from the client making an HTTP request to the backend API. FastAPI routes the request to the appropriate endpoint, where it is processed by the corresponding router function. The router function interacts with the database through SQLAlchemy ORM, executing CRUD operations as necessary to fulfill the request. Once the database operations are complete, a response is generated and returned to the client.

Understanding this request handling flow is essential for developers to comprehend the inner workings of the backend system and troubleshoot any issues that may arise during development or deployment.

8.3.4 Project Structure

The backend part of the locker system project adheres to a well-organized structure, following best practices to ensure clarity, modularity, and maintainability. Below is an overview of the project structure and the rationale behind each component:

- **README.md**: This file serves as the project's main documentation hub, containing comprehensive information on setup instructions, deployment steps, and other essential details. A well-written README enhances project understanding and facilitates collaboration among team members.

- **requirements.txt**: This file enumerates all Python packages required by the project, making it easy for developers to install dependencies using pip. Managing dependencies in a centralized file promotes consistency and reproducibility across different environments.

- **.env**: Environment variables play a crucial role in configuring the application, especially sensitive information like database credentials. Utilizing a separate .env file allows for easy management of configuration settings across various deployment environments.

- **venv**: The virtual environment isolates project dependencies, ensuring compatibility and preventing conflicts with other Python projects. By encapsulating dependencies within a dedicated environment, developers can maintain a clean and reproducible development setup.

- **alembic/versions**: This directory houses database migration files managed by Alembic, a lightweight migration tool for SQLAlchemy. Organizing migrations in a structured manner facilitates version control and systematic evolution of the database schema over time.

- **images**: Images used in the project documentation, such as the README file, are stored in this directory. Including visual aids enhances readability and comprehension of project-related instructions and guidelines.

- **Dockerfile**: Docker simplifies application deployment by encapsulating the application and its dependencies into portable containers. The

Dockerfile specifies the steps to build a Docker image, promoting consistency and reproducibility across different deployment environments.

The **app** folder encapsulates the core components of the backend application. It is further divided into the following subdirectories and files (examples are used from the Category class, but the structure is the same for all classes):

- **api** folder: This folder contains endpoint-specific modules responsible for handling HTTP requests and responses. Each endpoint module typically consists of three main components:

  - **CRUD operations (Create, Read, Update, Delete)**: These operations define the basic functionalities for interacting with database entities. For example, the `crud.py` files contain functions for creating, retrieving, updating, and deleting records from the database.

    The following code snippet demonstrates the `delete_category_by_id` function, which is responsible for deleting a category from the database by its ID:

```
1         def delete_category_by_id(category_id:
  int, db: Session):
2             category = get_category_by_id(
  category_id, db)
3             if category:
4                 db.delete(category)
5                 db.commit()
6
```

    This function first retrieves the category with the specified ID using the `get_category_by_id` function. If the category exists,

it is deleted from the database using the SQLAlchemy `delete` method, followed by a commit to persist the changes.

– **Router functions**: The `router.py` files define FastAPI router instances responsible for routing HTTP requests to the appropriate endpoint functions. Routers enhance code organization and modularity by grouping related endpoint operations together.

The following code snippet illustrates a router function responsible for handling HTTP DELETE requests to delete a category:

```
1       @router.delete('/categories/{category_id}', response_model=None, tags=['category'])
2       def delete_category(
3               category_id: int,
4               db: Session = Depends(get_db)):
5           category_crud.delete_category_by_id(category_id, db)
6           return Response(status_code=status.HTTP_204_NO_CONTENT)
7
```

In this function, the route decorator specifies the HTTP method (DELETE) and the endpoint URL pattern ("/categories/category_id"). The function parameters include the category ID to be deleted and a database session dependency obtained through the `get_db` function. Inside the function, the `delete_category_by_id` function from the CRUD module (`category_crud`) is called to delete the category from the database. Finally, a 204 No Content response is returned to indicate successful deletion.

– **Schema definitions**: Schemas define the structure and validation rules for request and response payloads exchanged between the client and server. The `schemas.py` files contain Pydantic

models representing data schemas for serialization and validation purposes.

The following code snippet presents a Pydantic model (`CategoryBaseSchema`) defined in a schema file:

```
        class CategoryBaseSchema(BaseModel):
            name: str

```

In this schema definition, `CategoryBaseSchema` inherits from `BaseModel`, a Pydantic base class used for defining data models. The schema consists of a single field (`name`) with a data type of `str`, representing the name of a category. Pydantic models provide automatic data validation based on the specified field types, enabling robust input validation and serialization/deserialization of data between client and server components.

- **db** folder: This folder contains modules related to database management and configuration. Key components include:
  - **Configuration module (`config.py`)**: This module defines database connection settings using environment variables and provides functions for establishing database connections and checking connectivity status. Centralizing database configuration facilitates easy maintenance and deployment across different environments.

    The following code snippet illustrates the `config.py` module, which manages database configuration:

```
        from sqlalchemy import create_engine
        from sqlalchemy.orm import sessionmaker
        from sqlalchemy.exc import
    OperationalError
        from dotenv import load_dotenv
```

```
5       import os
6
7       load_dotenv()
8
9       DB_USERNAME = os.getenv("
    DATABASE_USERNAME")
10      DB_PASSWORD = os.getenv("
    DATABASE_PASSWORD")
11      DB_HOST = os.getenv("DATABASE_HOST")
12      DB_PORT = os.getenv("DATABASE_PORT")
13      DB_NAME = os.getenv("DATABASE_NAME")
14      DB_URL = f"postgresql://{DB_USERNAME}:{
    DB_PASSWORD}@{DB_HOST}:{DB_PORT}/{DB_NAME}"
15
16      def get_database_url():
17          return DB_URL
18
19      engine = create_engine(DB_URL)
20      SessionLocal = sessionmaker(autocommit=
    False, autoflush=False, bind=engine)
21
22      def check_database_connection():
23          try:
24              engine = create_engine(DB_URL)
25              with engine.connect():
26                  return True
27          except OperationalError:
28              return False
29
```

This module loads database connection settings from environment variables using the `dotenv` library. It defines functions for retrieving the database URL, establishing a database engine, and

checking database connectivity. By encapsulating database configuration in a single module, the codebase becomes more modular and maintainable, enabling seamless deployment across various environments.

– **Model definitions (`models.py`)**: Models define the structure of database tables and their relationships using SQLAlchemy declarative base. Models encapsulate business logic and data integrity constraints, promoting code organization and reusability.

  The following code snippet demonstrates a model definition (`Category`) in the `models.py` module:

```
1       from sqlalchemy import Column, Integer,
    String
2       from db.config import Base
3
4       class Category(Base):
5           __tablename__ = 'categories'
6
7           id = Column(Integer, primary_key=
    True)
8           name = Column(String, nullable=False
    )
9
```

  In this model definition, `Category` is a SQLAlchemy model representing the `categories` table in the database. It inherits from `Base`, the declarative base provided by SQLAlchemy. The `__tablename__` attribute specifies the table name, while `id` and `name` represent the table columns. By encapsulating database schema in model classes, the codebase becomes more organized and maintainable, facilitating data manipulation and ensuring data integrity.

- **main.py**: The main entry point of the backend application, responsible for initializing the FastAPI application instance and configuring middleware, routers, and other application-level settings.

8.4 Arduino

Written by: Felix

9 Testing / Evaluation (Ting)

1. how did we test?
2. what did we test?
3. evaluation of test results

10 Reflection / Lessons Learned (Sven)

what would we do differently if we could turn back time and WHY

11 Next steps / Outlook (Felix)

12 Summary Ifdi

13 Append

Appendix

References

[1] A. Einstein, "On the electrodynamics of moving bodies," *Annalen der Physik*, vol. 17, no. 10, pp. 891–921, 1905.

[2] C. Samiullah, "Ultimate fastapi tutorial pt.7: Sqlalchemy database setup." `https://christophergs.com/tutorials/ultimate-fastapi-tutorial-pt-7-sqlalchemy-database-setup/`, 2022. Accessed: 2024-05-09.