Group 14
Advanced Software Technik
Bou Kamel Nancy 106035, Ibrahim Rana Muhammad 106728, Kamal Md Mostafa 929903, Mishra Chiraag106073
https://github.com/Nancy123a/berlin_charging_station2

Introduction:

ChargeHub Berlin is an interactive platform designed to help Berliners easily find electric vehicle (EV) charging stations in their area. It allows users to check the availability and utilization of charging stations, and actively engage in improving the charging infrastructure. Users can search for charging stations based on their postal code, rate locations, report malfunctions, and suggest new charging stations. The platform aims to streamline EV charging experiences by providing real-time information and facilitating communication among users.

Key Features:

Charging Station Search by Postal Code

- Users can enter their postal code and find available charging stations within that area.
- Charging station availability is displayed clearly, using colors or text indicators.
- Map is zoomed automatically to the searched area.
- The information of the charging station is displayed as we click on the pin.
- Pin is colored according to the power charging device (Low Power -> green, Medium Power -> yellow, High Power -> orange, Ultra high power -> red).

Malfunction Reporting:

- Users can report any malfunctions or problems with the charging stations via a simple form.
- The app displays reported issues to other users to inform them about the status of the charging stations.
- Admin manage the report and assign it to the charging station operator
- Charging station operator resolve the malfunction and after fixing it he inform the user
- The report should be assigned to an admin or charging station operator only if they have fewer than 10 assigned reports; otherwise, it is assigned to the first admin or cs operator in the queue.

Problem Statement:

As Berlin's streets welcome more electric vehicles, the demand for reliable and accessible charging stations is surging. Yet, many drivers struggle with outdated information, unavailable spots, and uncertainty about station quality. Enter ChargeHub Berlin—a smart, community-driven app that makes finding and reviewing charging stations effortless. With real-time updates, user reviews, and quality assessments, ChargeHub Berlin transforms the way EV drivers power up, ensuring a seamless, stress-free charging experience while fostering a connected and informed community.

Project Objective:

- **Enhance Charging Station Accessibility**: Enable users to quickly find and navigate to available charging stations using a postal code search and interactive map.

- **Improve User Convenience:** Provide real-time availability updates with clear indicators and a seamless interface for locating charging stations.
- **Ensure Transparent Charging Information:** Offer details about charging stations, including power levels and user reviews, to help EV drivers make informed decisions.
- **Facilitate Community Engagement:** Allow users to report malfunctions and share real-time station updates, fostering a collaborative ecosystem.
- **Optimize Maintenance and Issue Resolution:** Streamline the reporting and resolution process to minimize downtime of charging stations.
- **Ensure Fair Workload Distribution:** Implement a queue-based assignment system to balance report handling among admins and station operators.

Concrete Project Deliverables:

- **Postal Code Search Functionality:** Users can input their postal code to locate nearby charging stations.
- **Interactive Map with Charging Station Pins:** Stations are displayed on a map, with pins color-coded based on power level.
- **Real-Time Availability Indicators:** Charging station status is clearly shown using color or text indicators.
- **Station Detail View:** Clicking on a pin displays detailed information about the station.
- **Malfunction Reporting System:** A simple form allows users to report issues with charging stations.
- **Issue Display for Users:** Reported problems are visible to inform other users of station conditions.
- **Admin and Operator Assignment System:** Reports are assigned dynamically, ensuring even workload distribution.
- **Resolution and Feedback Mechanism:** Operators can update the status of resolved issues, notifying the reporting user.

Technology Stack:

- **Communication:** Whatsapp group MS Team - team collaboration. Updates brainstorming.
- **Modeling:** Miroboard (User case diagram, DDD, TDD, Event storming, UML) - brainstorming and visualizing domain models and workflows.
- **Programming languages:** Python: Python for versatility, ease of development, and strong ecosystem for backend and data processing.
- **Front-end technology:** streamlit - simplicity and efficiency in building interactive web applications. It allows for rapid prototyping, seamless integration with Python, and an intuitive UI for displaying real-time charging station data and reports without the need for complex front-end development.
- **IDE:** Jupyter lab - its interactive environment, making it ideal for data analysis, prototyping, and debugging. It allows you to test code, visualize data, and iterate quickly while developing features for the charging station app.
- **Database:** sqlalchemy:SQLAlchemy provides an efficient and flexible ORM (Object-Relational Mapping) for interacting with databases, ensuring smooth data management, scalability, and simplified query handling.
- **Testing:** pytest: its simplicity, scalability, and powerful testing capabilities. It makes writing, organizing, and running tests efficient, ensuring the reliability and stability of your application.
- **Version Control:** Github - manage source code and collaboration.

Project Development Documentation:

1. User Case Diagram

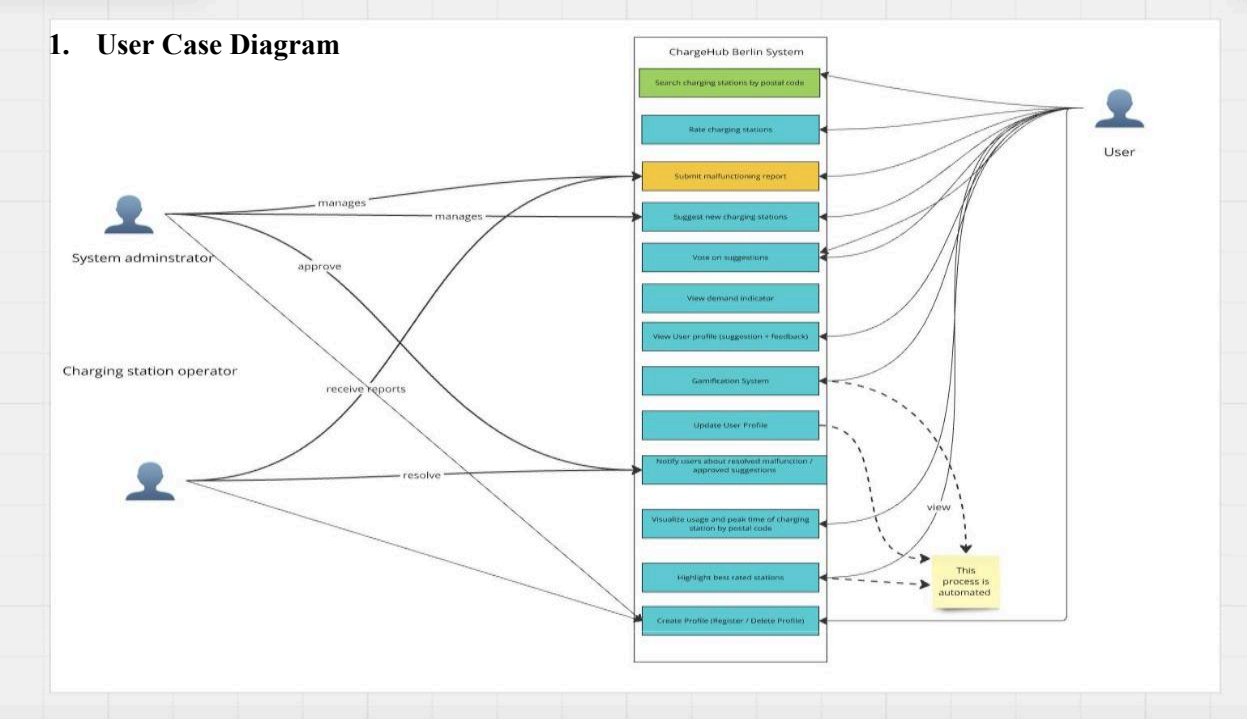


Fig 1: User Case Diagram of Berlin Hub

The ChargeHub Berlin system is accessed by three main roles: Users, System Administrators, and Charging Station Operators. Users can search for charging stations, submit malfunction reports, view feedback, and engage in the gamification system. System Administrators oversee malfunction reports, suggest new stations, and approve user notifications for resolved issues and station proposals. Charging Station Operators receive malfunction reports, resolve issues, and update the system upon completion. Each role contributes to maintaining an efficient and user-friendly charging network.

Event Storming and Business rules:

Event storming flow helps visualize and streamline complex processes by mapping out events, interactions, and dependencies in a collaborative way. The password must meet the following criteria: it should contain at least one numeric digit, one special character, one lowercase letter, one uppercase letter, and be at least eight characters long. The postal code must be numeric, exactly five digits long, and start with 10, 12, 13, or 14 to ensure it belongs to the Berlin area. Users, Administrators, and Charging Station Operators must have a unique username when creating an account.

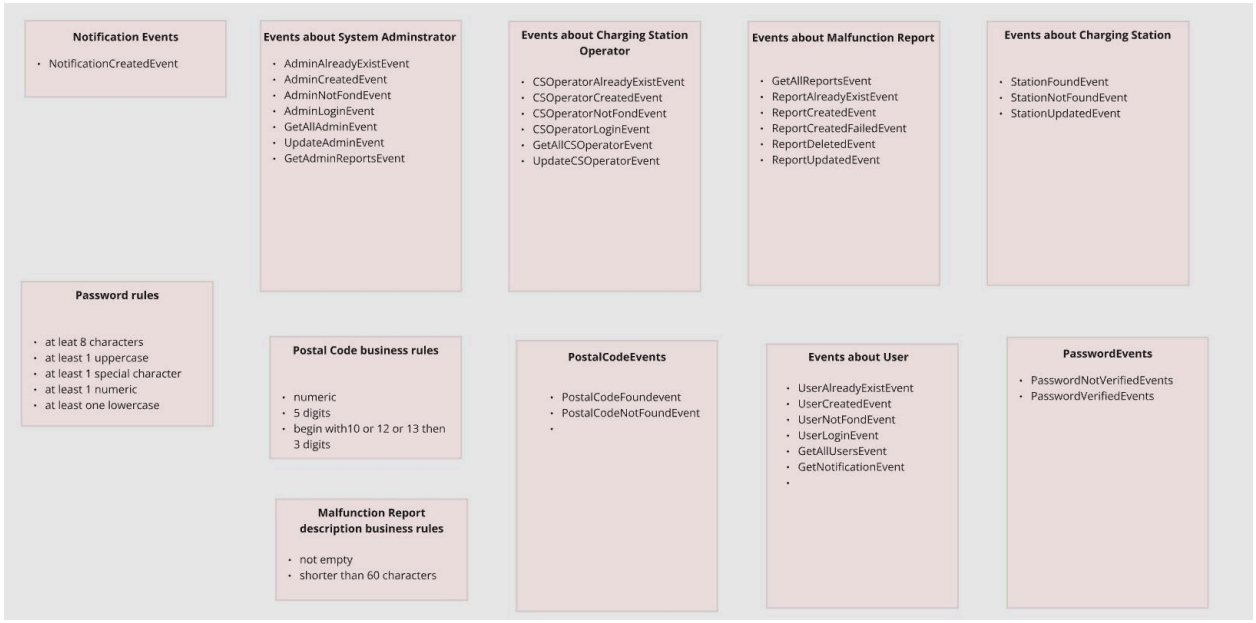


Fig2: Event Storming of Berlin Hub

2. UML

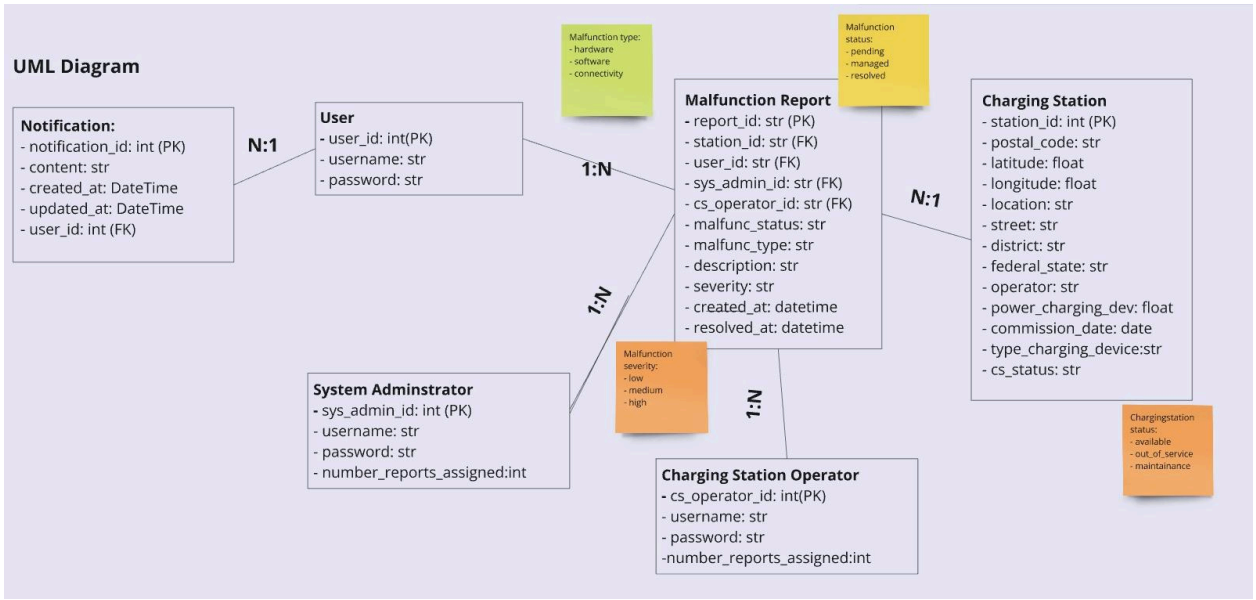


Fig 3: UML Diagram

Figure 3 illustrates the UML Diagram of ChargeHub Berlin. The relationship between notifications and users is N to 1, meaning each notification is assigned to one user, and each user receives one notification. Similarly, the relationship between malfunction reports and charging stations follows an N to 1 structure, where each charging station receives multiple reports, but each report belongs to a single station. On the other hand, the relationship between Users, System Administrators, and Charging Station Operators with malfunction reports is 1 to N, meaning each malfunction report is managed by one admin or operator, but each admin or operator can handle multiple reports. This design ensures clear responsibility assignment,

preventing confusion by linking each notification and malfunction report to a single recipient. It enables efficient report handling, allowing admins and operators to manage multiple reports while ensuring accountability.

```python
from sqlalchemy import Column, Integer, String, Float
from database.database import engine,SessionLocal,Base
from sqlalchemy.orm import relationship

# Define the User table
class User(Base):
    __tablename__ = 'user'  # Table name

    user_id = Column(Integer, primary_key=True, autoincrement=True)
    username = Column(String, unique=True, nullable=False)
    password = Column(String, nullable=False)

    reports = relationship("Report", back_populates="user")
    notifications = relationship("Notification", back_populates="user")
```

Fig4: User UML table implementation

```python
from sqlalchemy import Column, Integer, String
from database.database import Base  # Ensure Base is imported
from sqlalchemy.orm import relationship

class Admin(Base):
    __tablename__ = 'admin'  # Table name

    sys_admin_id = Column(Integer, primary_key=True, autoincrement=True)
    username = Column(String, unique=True, nullable=False)
    password = Column(String, nullable=False)
    number_reports_assigned = Column(Integer, nullable=False)

    reports = relationship("Report", back_populates="admin")
```

Fig 5: Admin UML table implementation

```python
from sqlalchemy import Column, Integer, String, Float
from database.database import engine, Base  # Adjust import path as needed
from sqlalchemy.orm import relationship

# Define the User table
class CSOperator(Base):
    __tablename__ = 'csoperators'  # Table name

    cs_operator_id = Column(Integer, primary_key=True, autoincrement=True)
    username = Column(String, unique=True, nullable=False)
    password = Column(String, nullable=False)
    number_reports_assigned = Column(Integer, nullable=False)

    reports = relationship("Report", back_populates="csoperator")
```

Fig 6: Charging Station Operator UML table implementation

Figures 4, 5 and 6 illustrate the Python implementation of the User, System Administrator, and Charging Station Operator UML tables respectively. The user_id, cs_operator_id, and sys_admin_id serve as

primary unique keys, ensuring each entity has a distinct identifier. Additionally, username is unique to prevent duplicate accounts with the same username.

```python
from sqlalchemy import Column, Integer, String, Float, Date
from sqlalchemy.ext.declarative import declarative_base
from database.database import Base  # Adjust import path as needed
from sqlalchemy.orm import relationship

class ChargingStation(Base):
    __tablename__ = "chargingstation"

    station_id = Column(Integer, primary_key=True, index=True)
    postal_code = Column(String, index=True)
    latitude = Column(Float)
    longitude = Column(Float)
    location = Column(String)
    street = Column(String)
    district = Column(String)
    federal_state = Column(String)
    operator = Column(String)
    power_charging_dev = Column(Float)
    commission_date = Column(Date)
    type_charging_device = Column(String)
    cs_status = Column(String)

    reports = relationship("Report", back_populates="chargingstation")
```

Fig 7: Charging Station UML table implementation

```python
from sqlalchemy import Column, Integer, String, DateTime, Enum, func, ForeignKey
from database.database import Base
from sqlalchemy.orm import relationship

# Define the Report table
class Report(Base):
    __tablename__ = 'report'  # Table name

    report_id = Column(Integer, primary_key=True, autoincrement=True)
    status = Column(Enum('pending', 'managed', 'resolved'), nullable=False, default='pending')
    description = Column(String, nullable=False)
    severity = Column(Enum('low', 'medium', 'high'), nullable=False, default='low')
    type = Column(Enum('hardware', 'software', 'connectivity'), nullable=False, default='hardware')
    created_at = Column(DateTime, default=func.now(), nullable=False)
    updated_at = Column(DateTime, default=func.now(), onupdate=func.now(), nullable=False)

    station_id = Column(Integer, ForeignKey('chargingstation.station_id'))
    user_id = Column(Integer, ForeignKey('user.user_id'))
    admin_id = Column(Integer, ForeignKey('admin.sys_admin_id'))
    csoperator_id = Column(Integer, ForeignKey('csoperators.cs_operator_id'))

    user = relationship("User", back_populates="reports")
    admin = relationship("Admin", back_populates="reports")
    csoperator = relationship("CSOperator", back_populates="reports")
    chargingstation = relationship("ChargingStation", back_populates="reports")
```

Fig 8: Malfunction report UML table implementation

```python
from sqlalchemy import Column, Integer, String, DateTime, Enum, func, ForeignKey, Boolean
from database.database import Base
from sqlalchemy.orm import relationship

# Define the Notification table
class Notification(Base):
    __tablename__ = 'notification'  # Table name

    notification_id = Column(Integer, primary_key=True, autoincrement=True)
    content = Column(String, nullable=False)
    created_at = Column(DateTime, default=func.now(), nullable=False)
    updated_at = Column(DateTime, default=func.now(), onupdate=func.now(), nullable=False)

    user_id = Column(Integer, ForeignKey('user.user_id'))
    user = relationship("User", back_populates="notifications")
```

Fig 9: Notification UML table implementation

Domain Event Flow

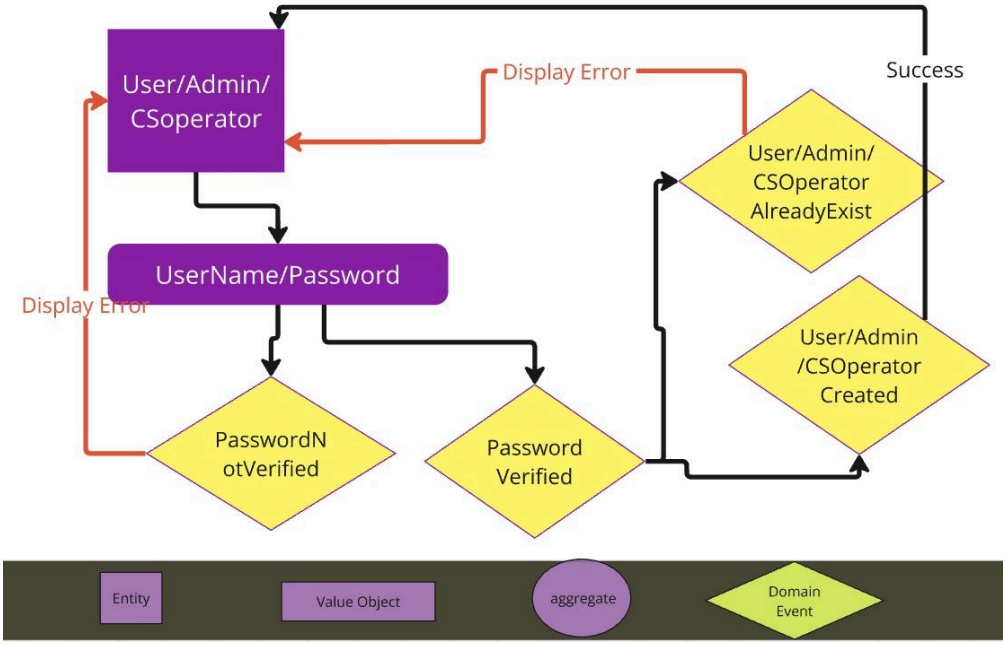


Fig 10: Domain Event Flow of Signup

Figure 10 illustrates the domain event flow of Charging Station Operator SignUp. User Admin and Charging Station Operator are entities, meaning they have a unique identity that persists over time, while username and password are value objects, defined by their attributes rather than a unique identity. If a user, admin, or charging station operator enters an invalid password during signup, a PasswordNotVerified event is triggered. If the password is valid but the user already exists, a UserAlreadyExistEvent is returned. However, if the password is verified and the user account does not exist, a UserCreatedEvent is generated.

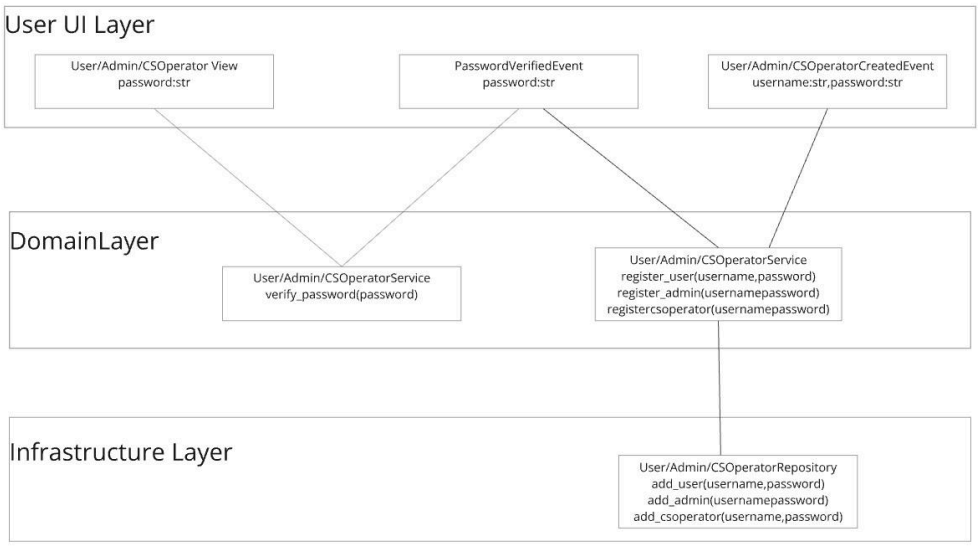


Figure 11: Domain Event Flow User, Admin, CSOperator Signup

Figure 11 illustrates the domain event flow of User, Admin, and Charging Station Operator signup. The user interface layer contains the view of the User and events such as PasswordVerifiedEvent and UserCreatedEvent. The domain layer includes services like UserService, which communicates with the infrastructure layer, where repositories, such as UserRepository, are responsible for data management and persistence. After the password is verified in the domain layer through the UserService, the PasswordVerifiedEvent is returned. If the password is verified, the register_user function in UserService is called, which in turn calls the add_user function in the UserRepository to add the user to the database. Finally, the UserCreatedEvent is returned to the user interface upon successful signup.

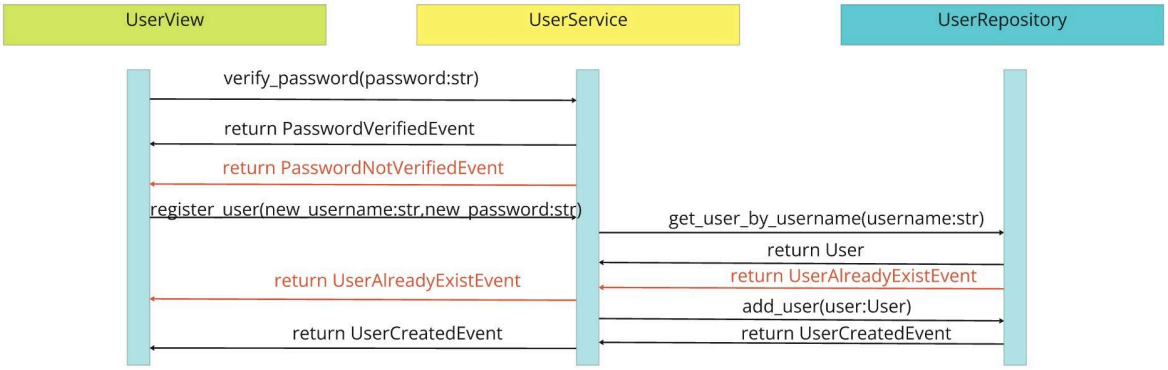


Fig 12: Sequence Diagram of User SignUp

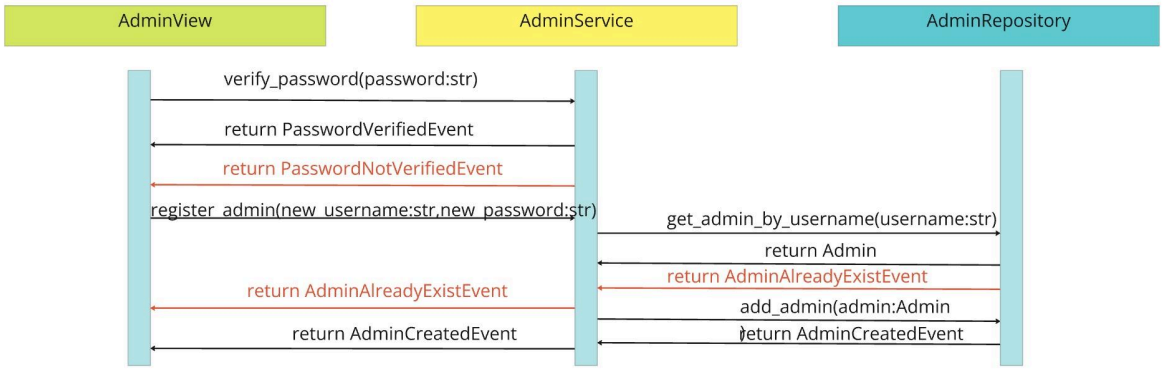


Fig 13: Sequence Event Diagram of Amin Signup

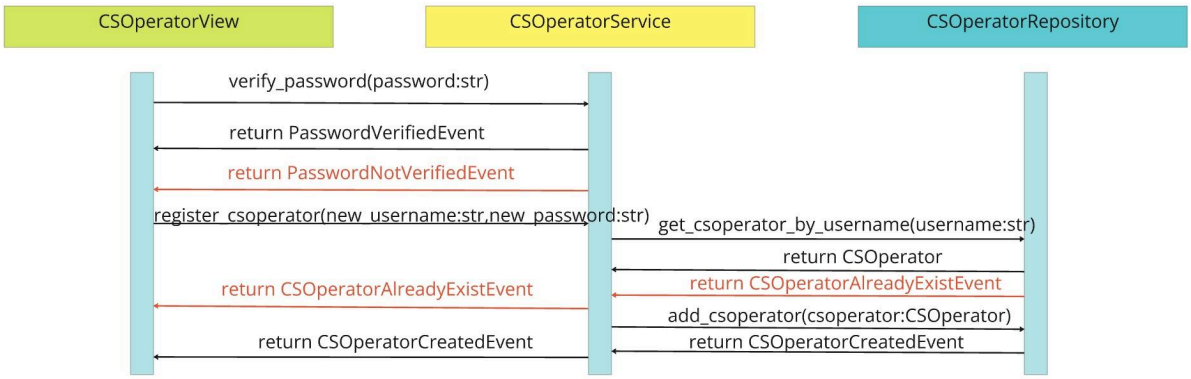


Figure 14: Sequence Event Diagram of CSOperator Signup

Figures 12, 13, and 14 show the sequence event diagrams for the signin process of User, Admin, and Charging Station Operator, respectively. The red arrows represent failed requests, such as password not verified or user already existing, while the black arrows indicate successful requests. These sequence event diagrams illustrate the communication between the user interface, domain, and infrastructure layers, as well as the flow of events. The UserView sends a verify_password request to UserService, which returns PasswordVerifiedEvent if the password is valid, or PasswordNotVerifiedEvent if not. The UserView then calls the register_user function in UserService, which checks if the user already exists by calling the get_user_by_username function. If the user already exists, UserAlreadyExistEvent is returned; otherwise, the function returns the user. If the user doesn't exist, UserService calls the add_user function in UserRepository, and a UserCreatedEvent is returned to both UserView and UserService.

Figure 15 shows the domain event flow for the signin process of User, Admin, and Charging Station Operator. Similar to Figure 11, the design consists of three layers: User interface, domain, and infrastructure. Upon receiving the PasswordVerifiedEvent, the signin_user function is called in UserService, which then calls the login_user function in UserRepository. This triggers the UserLoginEvent in the user interface. Figure 16 further illustrates the domain event flow for signin. If the password is verified and the username exists, a UserLoginEvent is triggered.

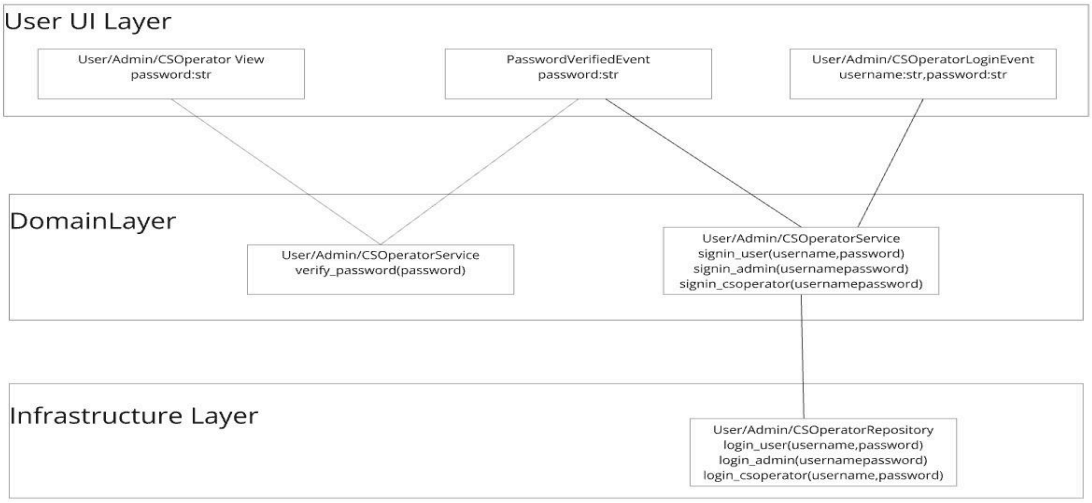


Figure 15: Domain event flow User, Admin, CSOperator Sign In

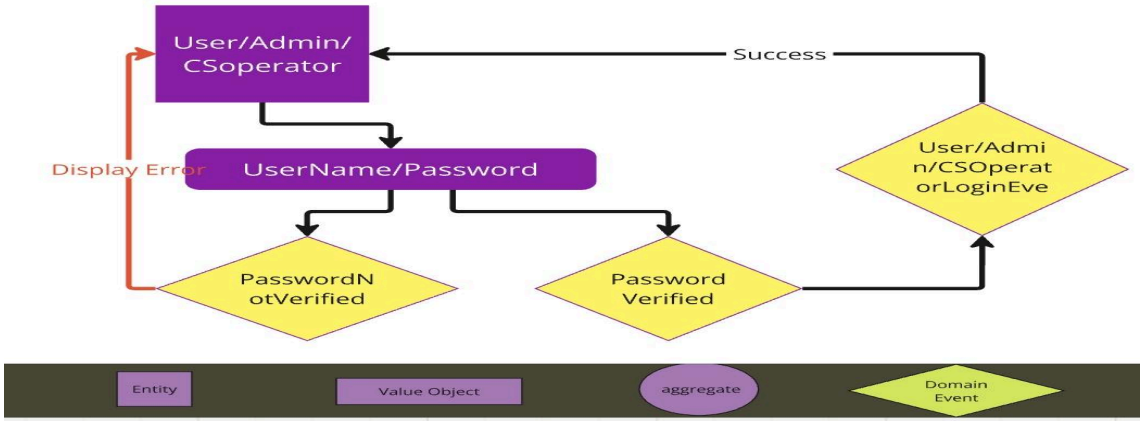


Fig 16: Domain event flow signin

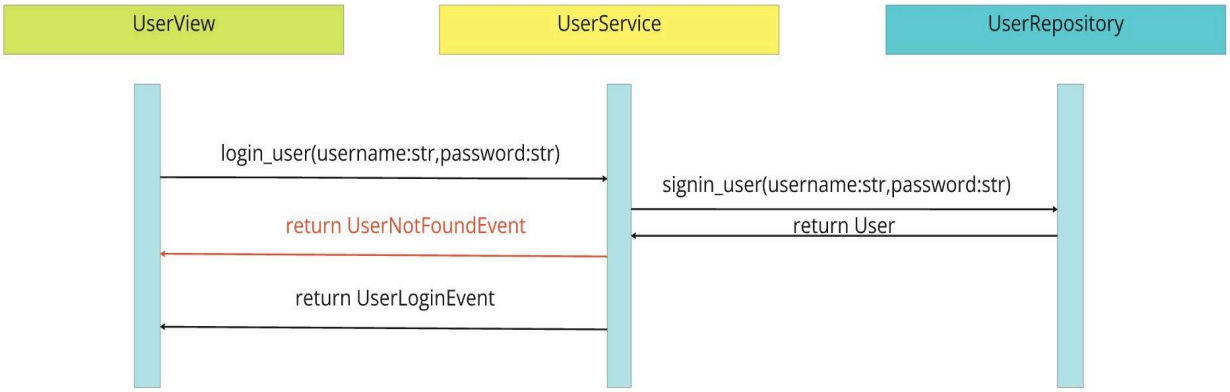


Figure 16: Sequence event diagram of user sign in

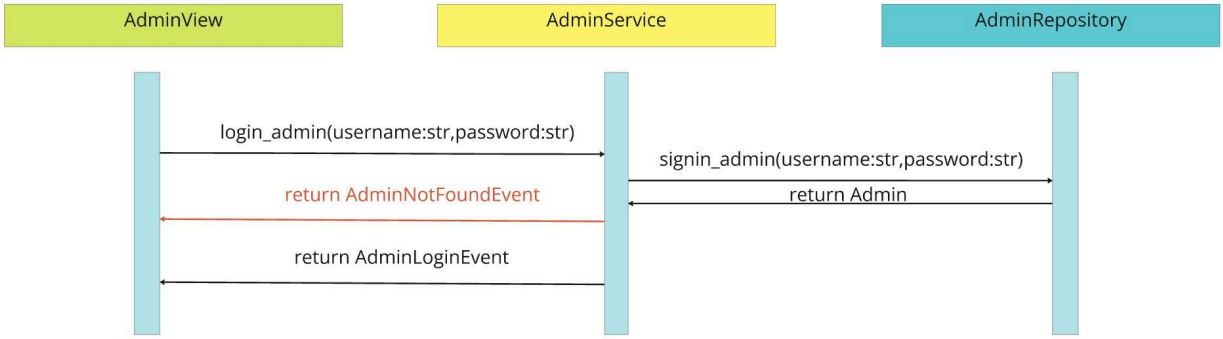


Figure 17: Sequence event diagram of Admin Sign in

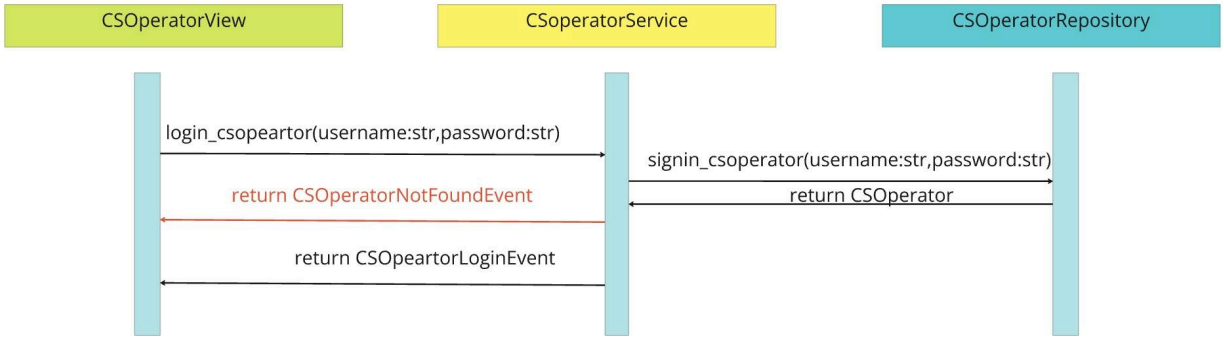


Figure 18: Sequence event diagram of charging station operator sign in

Figure 19 illustrates the domain event flow for User, Admin, and Charging Station Operator when searching for a charging station by postal code. ChargingStations is an aggregate that contains a list of charging stations. If the postal code is verified, PostalCodeFoundEvent will be triggered. If charging stations exist within the given postal code, the StationFoundEvent is triggered, displaying the charging stations on the map. If no stations are found, the StationNotFoundEvent is triggered, showing an error message in the user view.

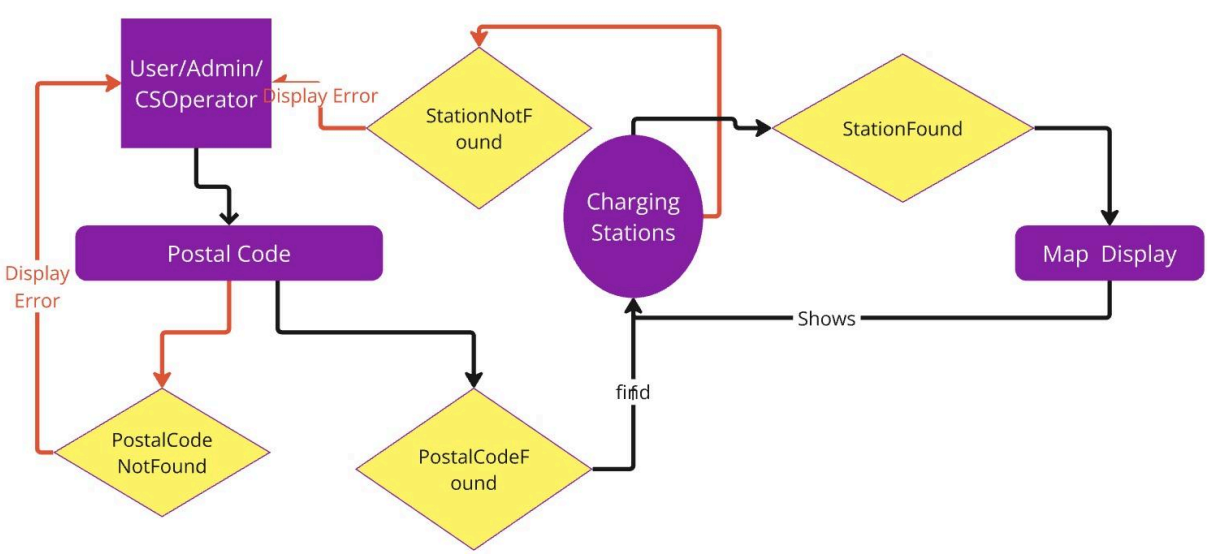


Figure 19: Domain Event flow of User Admin, CSOperator search charging station

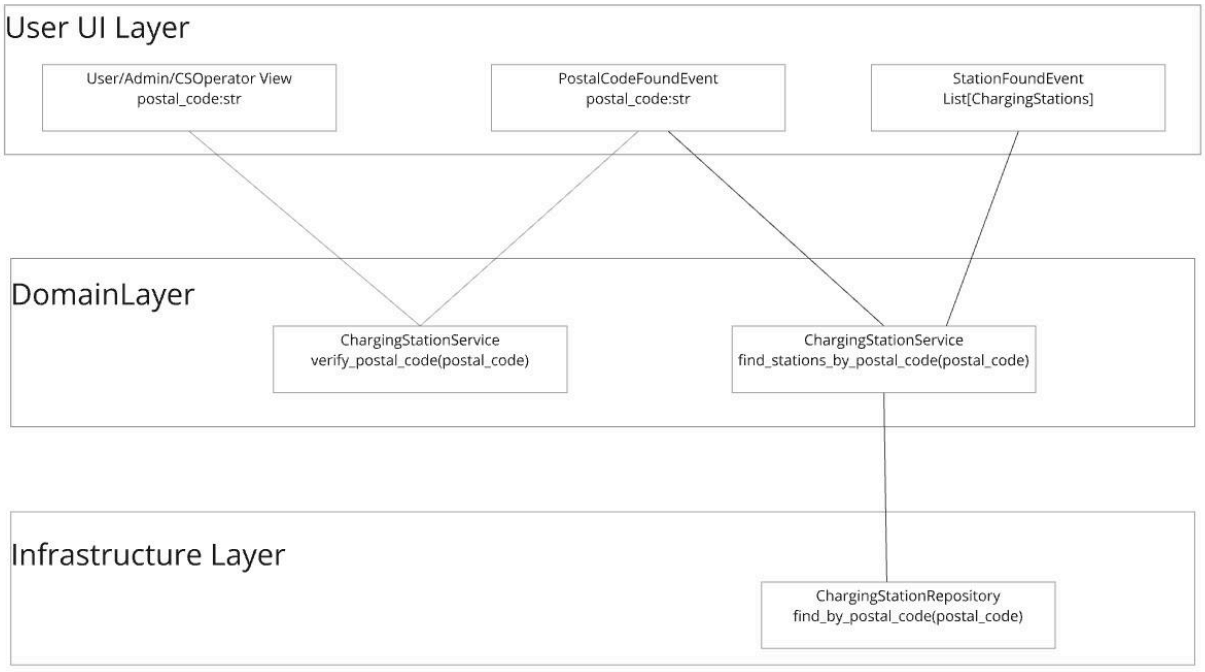


Figure 20: Domain event flow of search charging station by postal code

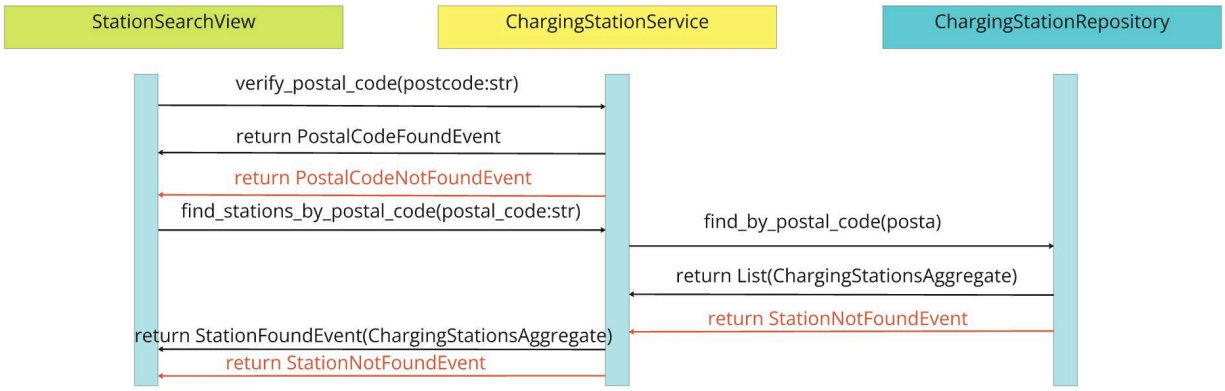


Figure 21: Sequence event diagram of search charging station by postal code

Figure 20 and Figure 21 show the domain event flow and sequence event diagram for searching charging stations by postal code. The process involves three layers: StationSearchView, ChargingStationService, and ChargingStationRepository. The StationSearchView calls the verify_postal_code function in ChargingStationService, which returns a PostalCodeFoundEvent if the postal code is valid, or a PostalCodeNotFoundEvent if it is not. If the postal code is found, the StationSearchView then calls find_station_by_postal_code in ChargingStationService, which in turn calls the find_by_postal_code function in ChargingStationRepository. If charging stations are available, a list of ChargingStations is returned; otherwise, the StationNotFoundEvent is triggered.

Domain Event Flow of Malfunction Reporting:

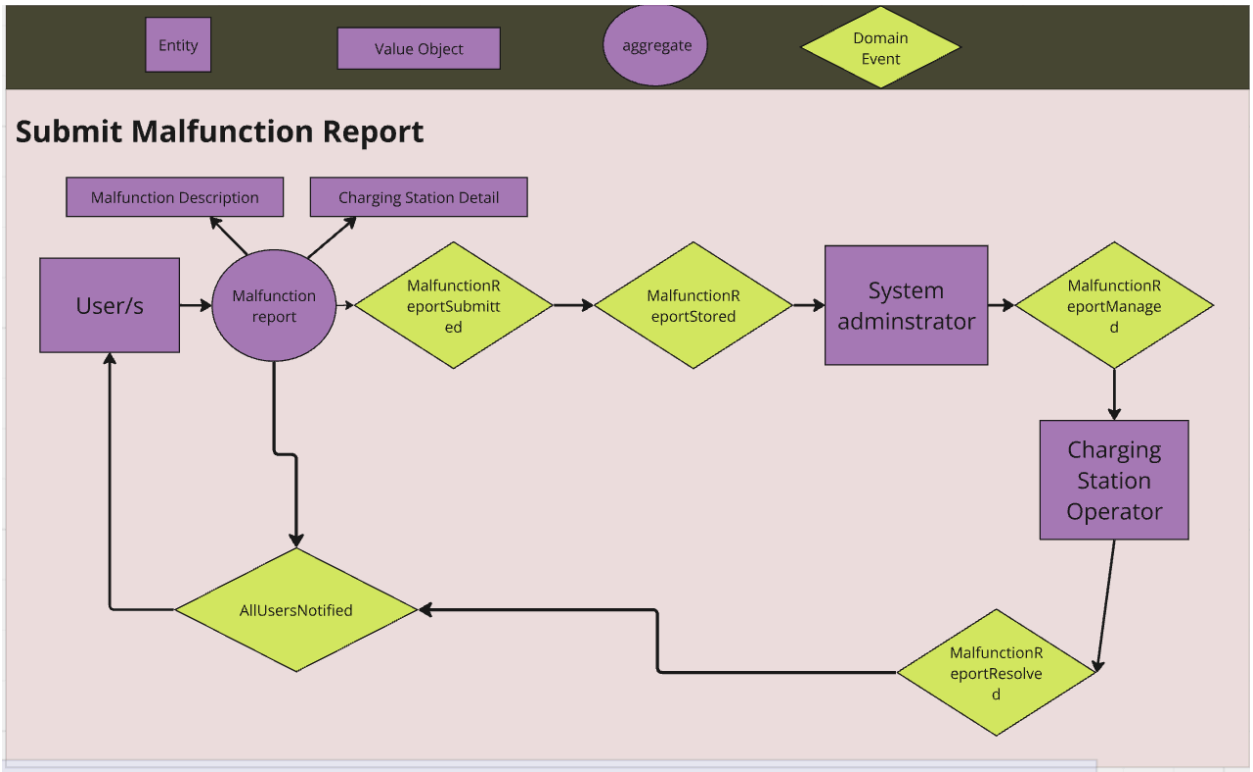


Fig 22: Domain Event Flow of Malfunction Reporting

Figure 22 illustrates the domain event flow of Malfunction Reporting. The primary entities in this process are the User, System Administrator, and Charging Station Operator, while the Malfunction Description and Charging Station Details serve as value objects, defined by their attributes rather than unique identity. The Malfunction Report acts as an aggregate, encapsulating these value objects and coordinating the domain events.

1. When a user submits a malfunction report, a MalfunctionReportSubmitted event is triggered. This event contains the malfunction details and the corresponding charging station information.
2. The system processes and stores the report, resulting in a MalfunctionReportStored event.
3. The System Administrator is then notified and begins managing the malfunction report. This action triggers a MalfunctionReportManaged event, signifying that the report is being handled.
4. The resolution task is assigned to a Charging Station Operator, who works on addressing the malfunction. Once the issue is resolved, a MalfunctionReportResolved event is generated.
5. To ensure transparency, the system notifies all users about the malfunction's resolution status through an AllUsersNotified event.

This domain event flow facilitates a systematic and efficient process for malfunction reporting, storage, management, and resolution, fostering accountability and seamless communication among the involved entities.

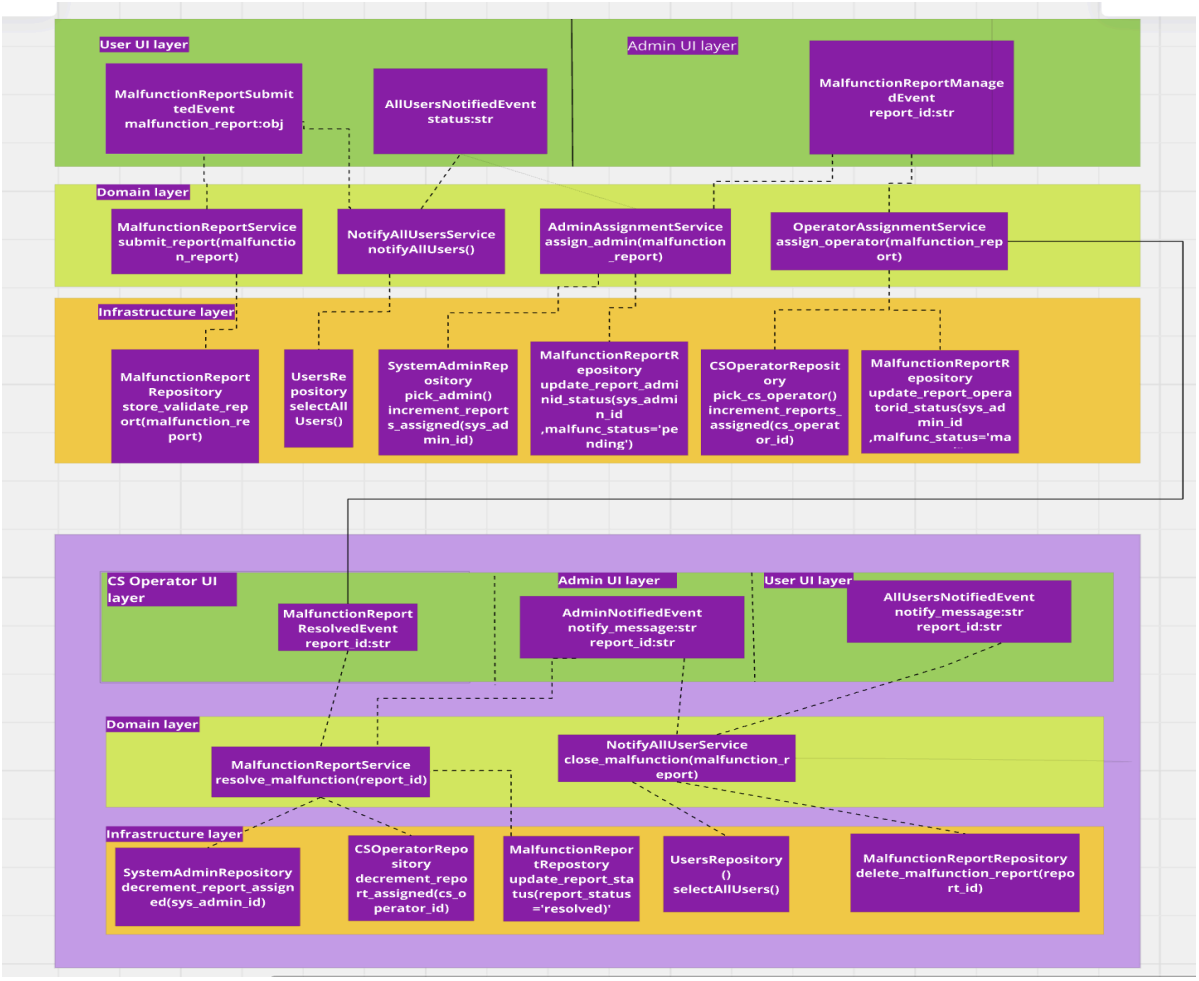


Fig 23: Domain Event Flow of User, Admin , and CSOperator in Malfunction Reporting

Figure 23 illustrates the domain event flow for malfunction reporting, highlighting interactions across the User UI layer, Admin UI layer, CS Operator UI layer, Domain layer, and Infrastructure layer.

- **User UI Layer:**
  - The process begins with the MalfunctionReportSubmittedEvent, where a user submits details of the malfunction.
  - The AllUsersNotifiedEvent is triggered to notify all users of the reported malfunction.
- **Domain Layer:**
  - The MalfunctionReportService handles the submission of the malfunction report and coordinates with the NotifyAllUsersService to propagate notifications to all users.
  - The AdminAssignmentService assigns an admin to the malfunction report, while the OperatorAssignmentService assigns a charging station operator.
- **Infrastructure Layer:**
  - The MalfunctionReportRepository is responsible for storing and validating malfunction reports, as well as updating their statuses (e.g., from "submitted" to "managed").
  - The UsersRepository retrieves the list of all users to notify them of the malfunction.
  - The SystemAdminRepository tracks admin assignments and increments their task counts.
  - The CSOperatorRepository selects a charging station operator and tracks their task assignments.
- **Admin UI Layer:**
  - The AdminNotifiedEvent updates the admin about the assigned malfunction report.
  - Once the malfunction is addressed, the MalfunctionReportManagedEvent is triggered, indicating that the issue has been assigned to an operator.
- **CS Operator UI Layer:**
  - The operator resolves the malfunction, triggering the MalfunctionReportResolvedEvent.
  - The resolution process involves the MalfunctionReportService, which updates the status of the malfunction to “resolved” through the MalfunctionReportRepository.
- **Final Steps:**
  - The NotifyAllUsersService generates the AllUsersNotifiedEvent to inform users of the resolution.
  - The MalfunctionReportRepository deletes the resolved malfunction report, ensuring the database remains clean.

This event flow ensures a clear delegation of responsibilities, seamless communication between layers, and transparency in handling malfunctions from submission to resolution.

Sequence event diagram of Malfunction Reporting and Resolving:

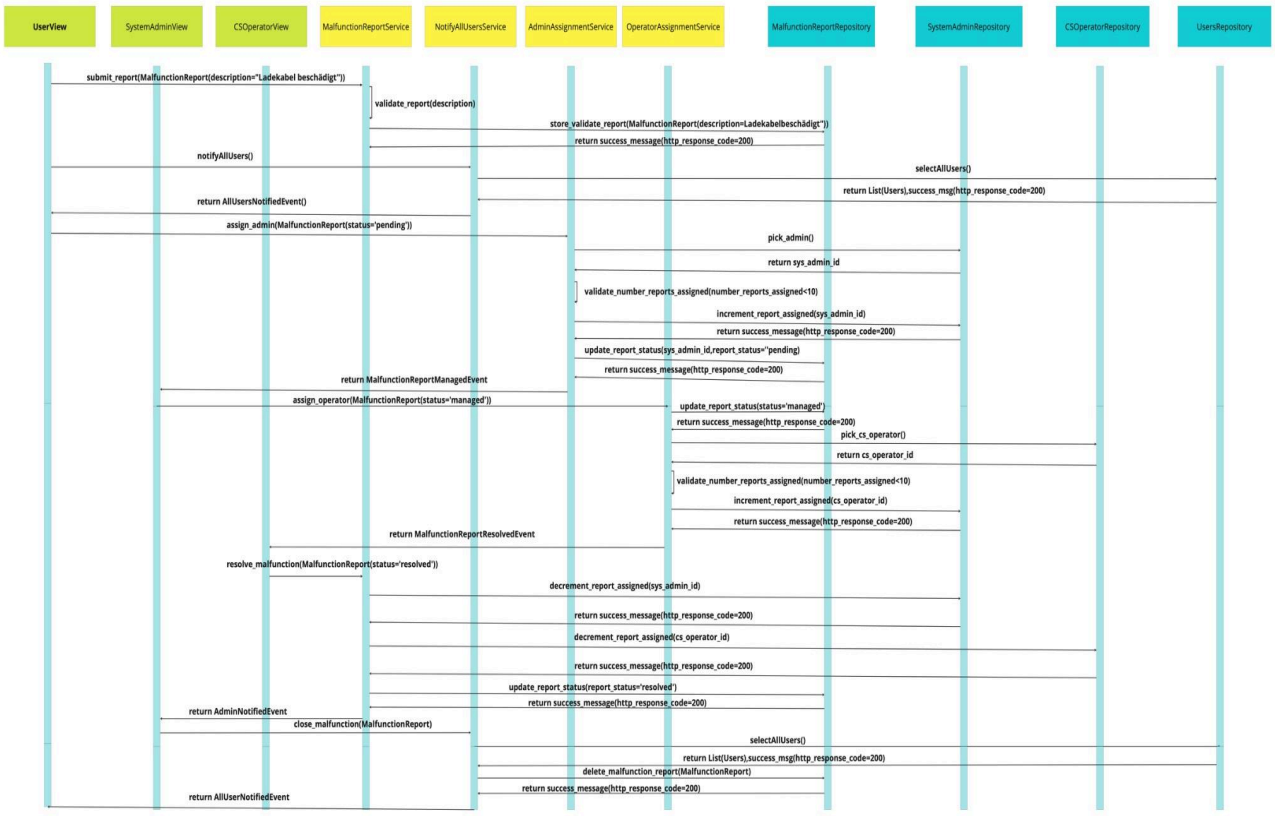


Figure 24: Sequence event diagram of Malfunction Reporting and Resolving

Figure 25 illustrates the sequential domain event flow for malfunction reporting, detailing interactions across UserView, AdminView, CSOperatorView, various domain services, and repositories.

- **Step 1 (User Interaction):**
  The process begins in the UserView, where a user submits a malfunction report using the submit_report(MalfunctionReport(description="Ladekabel beschädigt")) function. This initializes the malfunction report. The MalfunctionReportService validates the report using the validate_report(description) method, and the MalfunctionReportRepository stores the validated report through store_validate_report(), returning a success message (http_response_code=200). The NotifyAllUsersService is triggered, invoking the notifyAllUsers() function and generating the AllUsersNotifiedEvent to notify all users about the malfunction.

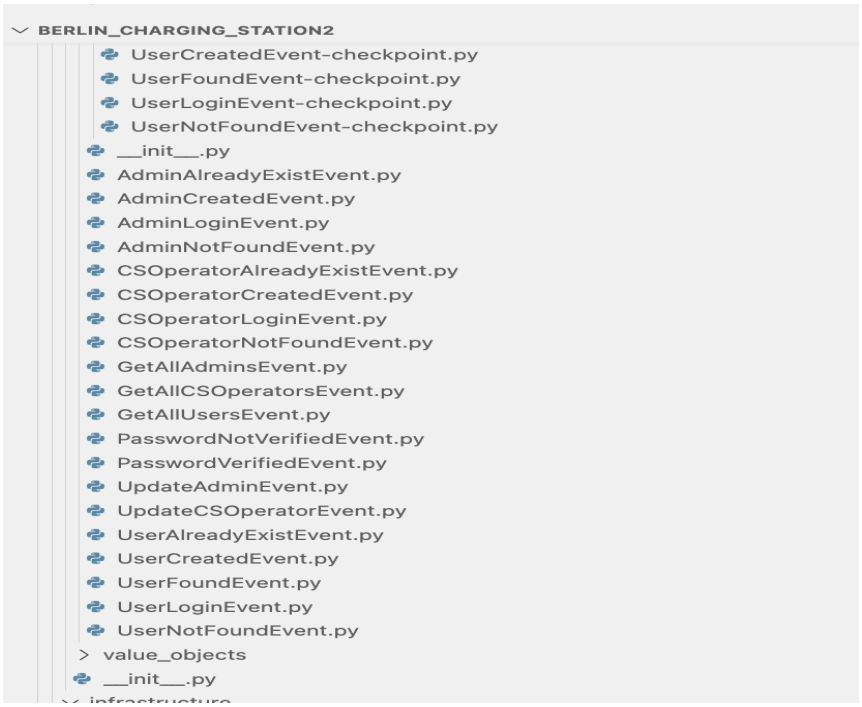
- **Step 2 (Admin Assignment):**
  The AdminAssignmentService assigns an admin to the malfunction report through the assign_admin(MalfunctionReport(status='pending')) method. The SystemAdminRepository picks an admin using the pick_admin() method, validates the number of reports assigned (validate_number_reports_assigned(number_reports_assigned<10)), and increments the admin's assigned report count using increment_reports_assigned(sys_admin_id), returning a success message (http_response_code=200). The MalfunctionReportRepository updates the report status (status='pending') and generates a success message, triggering the MalfunctionReportManagedEvent.

- **Step 3 (Operator Assignment):**
  The OperatorAssignmentService assigns a charging station operator through assign_operator(MalfunctionReport(status='managed')). The CSOperatorRepository selects an operator using pick_cs_operator(), validates the operator's workload (validate_number_reports_assigned(number_reports_assigned<10)), and increments the operator's assigned report count using increment_reports_assigned(cs_operator_id), returning a success message (http_response_code=200). The MalfunctionReportRepository updates the report status (status='managed') and returns a success message.

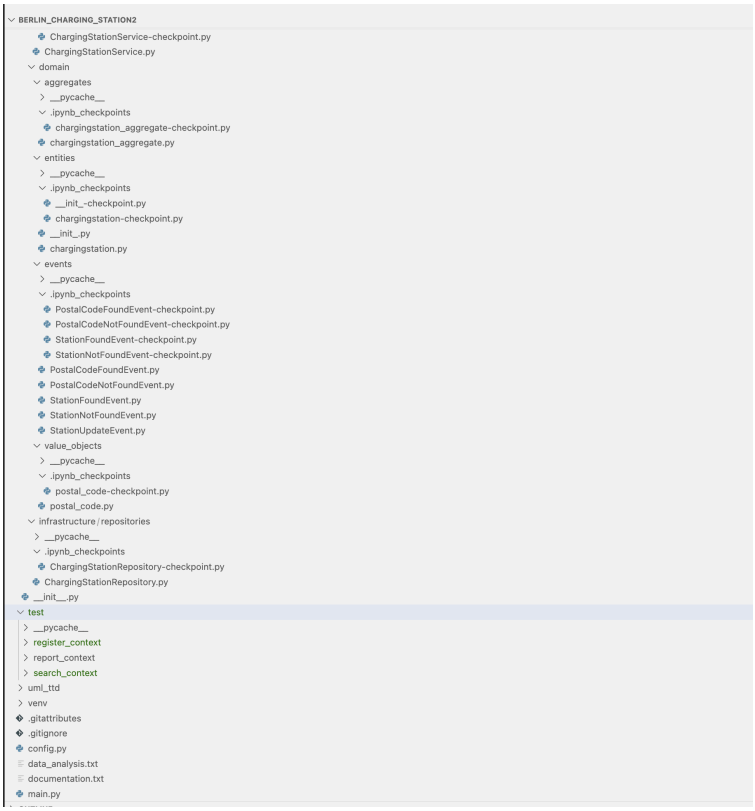
- **Step 4 (Resolution Process):**
  The operator resolves the malfunction using resolve_malfunction(MalfunctionReport(status='resolved')) in the MalfunctionReportService. The SystemAdminRepository and CSOperatorRepository decrement the assigned report counts for the respective admin and operator using decrement_reports_assigned() and return success messages (http_response_code=200). The MalfunctionReportRepository updates the report status to "resolved" and returns a success message. The MalfunctionReportResolvedEvent is triggered.

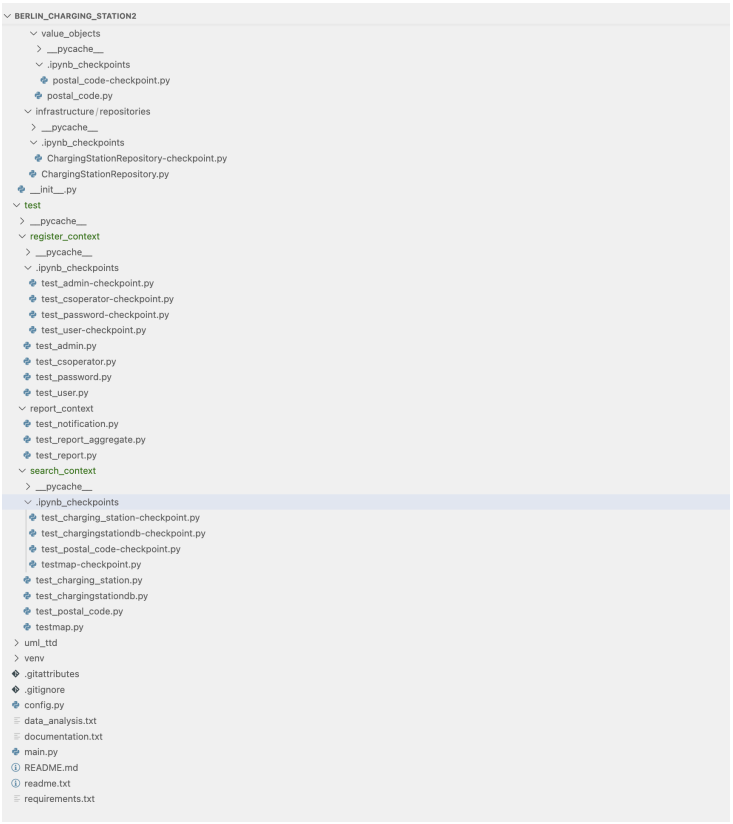
- **Step 5 (Notification and Cleanup):**
  The process concludes with the NotifyAllUsersService, which executes close_malfunction(MalfunctionReport). The UsersRepository retrieves all users using selectAllUsers()and returns a list of users and a success message. The MalfunctionReportRepository deletes the resolved malfunction report using delete_malfunction_report(report_id) and returns a success message (http_response_code=200). Finally, the AllUsersNotifiedEvent confirms that all users have been informed about the resolution.

Project Structure:




BERLIN_CHARGING_STATION2
__pycache__
NotificationService.py
ReportService.py
domain
aggregate
__pycache__
ReportAggregateService.py
entities
__pycache__
.ipynb_checkpoints
notification-checkpoint.py
report-checkpoint.py
notification.py
report.py
events
__pycache__
GetAdminReportsEvent.py
GetAllReportsEvent.py
GetUserNotificationsEvent.py
NotificationCreateEvent.py
ReportAlreadyExistsEvent.py
ReportCreateEvent.py
ReportCreateFailedEvent.py
ReportDeleteEvent.py
ReportUpdateEvent.py
value_objects
__pycache__
report_description.py
report_severity.py
report_type.py
infrastructure/repositories
__pycache__
NotificationRepository.py
ReportRepository.py
search_context
application/services
__pycache__
.ipynb_checkpoints
ChargingStationService-checkpoint.py
ChargingStationService.py
domain
aggregates
__pycache__
.ipynb_checkpoints
chargingstation_aggregate-checkpoint.py
chargingstation_aggregate.py
entities
__pycache__
.ipynb_checkpoints
__init_-checkpoint.py
chargingstation-checkpoint.py


BERLIN_CHARGING_STATION2
ChargingStationService-checkpoint.py
ChargingStationService.py
domain
aggregates
__pycache__
.ipynb_checkpoints
chargingstation_aggregate-checkpoint.py
chargingstation_aggregate.py
entities
__pycache__
.ipynb_checkpoints
__init_-checkpoint.py
chargingstation-checkpoint.py
__init_.py
chargingstation.py
events
__pycache__
.ipynb_checkpoints
PostalCodeFoundEvent-checkpoint.py
PostalCodeNotFoundEvent-checkpoint.py
StationFoundEvent-checkpoint.py
StationNotFoundEvent-checkpoint.py
PostalCodeFoundEvent.py
PostalCodeNotFoundEvent.py
StationFoundEvent.py
StationNotFoundEvent.py
StationUpdateEvent.py
value_objects
__pycache__
.ipynb_checkpoints
postal_code-checkpoint.py
postal_code.py
infrastructure/repositories
__pycache__
.ipynb_checkpoints
ChargingStationRepository-checkpoint.py
ChargingStationRepository.py
__init__.py
test
__pycache__
register_context
report_context
search_context
uml_ttd
venv
.gitattributes
.gitignore
config.py
data_analysis.txt
documentation.txt
main.py



This project is organized to reflect modularity and maintain a clear separation of responsibilities, enabling easy scalability and debugging. The core folder contains helper methods and essential backend utilities for interacting with different components of the application. The database folder manages database configurations and includes scripts for initializing and importing data, with subfolders like dbtables for further organization. The datasets folder stores raw CSV files, such as Ladesaeulenregister.csv, which serve as the foundation for data-driven operations.
The src directory encapsulates the primary application logic, categorized into contexts like register_context, search_context, and report_context, ensuring functionality is logically separated. Additionally, the test folder houses test cases for verifying application reliability across these contexts. The presence of structured configuration files like config.py and the entry point main.py ensures seamless initialization and integration of all layers of the project.

Explanation of Folder Content:

- Each user case is assigned a folder in the src folder: search_context, report_context, register_context
- The /domain folder contains the core business logic, including aggregates (e.g., ChargingStationAggregate), entities (e.g., User, Admin), value objects (e.g., Password, postal_code), and events (e.g., PostalCodeFoundEvent, UserCreatedEvent). It handles the system's essential rules and behaviors, ensuring consistency and managing the flow of data between entities, events, and value objects.
- The /application/services folder contains services for specific use cases (e.g., UserService, ChargingStationService). These services coordinate business logic by interacting with the domain layer, managing operations like user registration, charging station search, and event handling, to fulfill application requirements.

- The /infrastructure/repositories folder contains repositories for specific use cases (e.g., UserRepository, ChargingStationRepository). These repositories handle data persistence, interacting with the database to store, retrieve, and manage domain entities, ensuring data consistency and access for the application.
- The /core folder contains register_methods.py, which creates the login and registration Streamlit forms and manages user interface interactions with the domain layer for login and signup. Additionally, methods.py implements the Streamlit interfaces for searching charging stations and reporting malfunctions, enabling communication with the domain layer to search for stations or store malfunction reports.
- The /database contain databas.py file which sets up a SQLite database connection using SQLAlchemy. It creates the hubberlin.db file, establishes a session factory (SessionLocal) for database interactions, and defines Base as the base class for models.
- The /tests contain test cases for the 3 user cases: registration , search station, and report malfunction. It verify functionality for services and infrastructure
- The /datasets folder contains csv files for charging stations, residents etc…
- The main.py is the main entry point

Test Driven Development (TDD):

The project began with the creation of a GitHub repository, which was shared with the team to foster collaboration from the start. To kick off the development, we wrote initial test code to confirm the successful creation of the database, ensuring we were on the right track. We then used Streamlit to test the functionality of the app and make sure everything was working seamlessly.
As the project progressed, we refined our Domain-Driven Design (DDD) structure and integrated testing into the development workflow. This allowed us to consistently follow the Test-Driven Development (TDD) approach, ensuring that our code was always tested before it was fully implemented. TDD not only helped us maintain high-quality code but also made the debugging process more efficient and effective.
To achieve a test coverage of around 80%, we adopted pytest as our primary testing framework. The feedback from pytest gave us valuable insights into areas that needed more attention, prompting us to refine our tests and improve coverage. This iterative testing and refinement process ensured that we were thoroughly covering all parts of the application, ultimately leading to a more robust and reliable product.

1. **Focusing on Critical Components:** The critical components of my system included the following:

- **Postal Code Validation:** This functionality was essential to ensure that users input valid postal codes, and it directly impacted the correctness of the application.

- **Charging Station Data:** The core business logic around charging stations, including their attributes, status, and location, required thorough testing to ensure accuracy and functionality.

- **User Interface for Map Interaction:** The feature that displayed charging stations on a map using folium needed testing to ensure the correct markers were added, the map zoom worked as expected, and the color-coding of power levels was accurate.

- **Report Submission Validation:** Tests were implemented to ensure that all required fields, such as malfunction description and charging station details, were provided before a report could be submitted. Invalid or incomplete submissions were rejected, maintaining data integrity.

- **Notification Mechanism:** Test cases were written to verify that users, system administrators, and charging station operators received appropriate notifications at each stage of the malfunction reporting process. This ensured real-time communication throughout the workflow.

- **Role-Based Workflow Testing:** The process of assigning malfunction reports to system administrators and charging station operators was tested thoroughly. Test cases ensured that only authorized roles could access and manage specific tasks.

- **Status Updates:** Tests validated that the status of a malfunction report (e.g., "Submitted," "Stored," "Managed," "Resolved") was updated accurately at each step of the workflow. These updates were reflected both in the database and the user interface.

- **Error Handling:** Edge cases, such as duplicate malfunction reports or invalid charging station references, were tested to ensure the system handled errors gracefully without impacting overall functionality.

- **Database Interaction:** CRUD operations for malfunction reports were tested extensively, including adding new reports, retrieving report details by ID, updating statuses, and deleting resolved or invalid reports.

- **Integration with User Notifications:** Tests confirmed that notifications triggered by backend events, such as report resolution or rejection, were successfully sent to all relevant users.

- **End-to-End Testing:** The entire workflow, from report submission to resolution, was tested to ensure the system handled user input, database updates, and frontend notifications seamlessly.

2. **Test Coverage in Action:** I achieved 80% test coverage through the following testing strategy:

Unit Tests: I wrote unit tests for individual components, like validating the format of postal codes or ensuring that each charging station's data was stored correctly. For example, the tests for postal code validation (e.g., test_valid_postal_code and test_postal_code_invalid_format) ensured the postal code handling logic was correct.

```python
def test_valid_postal_code():
    """Test creation of a valid postal code."""
    postal_code = PostalCode(value="10115")
    assert postal_code.value == "10115"
    assert postal_code.is_valid()

def test_postal_code_invalid_format():
    """Test invalid postal code formats."""
    with pytest.raises(TypeError, match="Invalid postal code: 15123"):
        PostalCode(value="15123")  # Invalid due to prefix

    with pytest.raises(TypeError, match="Invalid postal code: 1000"):
        PostalCode(value="1000")  # Too few digits

    with pytest.raises(TypeError, match="Invalid postal code: 101151"):
        PostalCode(value="101151")  # Too many digits
```

Fig 22: test valid and invalid postal code

Integration Tests: I also wrote integration tests for components working together, such as ensuring that charging station data could be added to the database and rendered on the map (e.g., test_charging_stations_data_display). This also included verifying that power levels were color-coded appropriately based on the charging station's capacity (e.g., test_color_categorization_for_power).

```python
# Test 1: Test Charging Stations Data Display
def test_charging_stations_data_display(db_session):
    """Test that when a valid postal code is entered, the map displays the correct markers for charging stations."""
    # Create some test charging stations in multiple postal codes
    charging_station_1 = ChargingStation(
        station_id=123, postal_code="10001", latitude=40.7128, longitude=-74.0060, location="New York, USA",
        street="5th Avenue", district="Manhattan", federal_state="NY", operator="NYC Charging",
        power_charging_dev=50, commission_date=datetime.strptime("11.10.2020", "%d.%m.%Y").date(), type_charging_device="Fast", cs_status="Active"
    )
    charging_station_2 = ChargingStation(
        station_id=126, postal_code="10001", latitude=40.730610, longitude=-73.935242, location="New York, USA",
        street="Broadway", district="Manhattan", federal_state="NY", operator="NYC Charging",
        power_charging_dev=100, commission_date=datetime.strptime("11.10.2020", "%d.%m.%Y").date(), type_charging_device="Fast", cs_status="Active"
    )
    db_session.add(charging_station_1)
    db_session.add(charging_station_2)
    db_session.commit()

    # Generate map and add markers
    m = folium.Map(location=[40.7128, -74.0060], zoom_start=13)
    marker_cluster = MarkerCluster().add_to(m)

    # Adding markers to map
    for station in [charging_station_1, charging_station_2]:
        folium.Marker(
            location=[station.latitude, station.longitude],
            popup=f"Station ID: {station.station_id}, Operator: {station.operator}, Power: {station.power_charging_dev}kW"
        ).add_to(marker_cluster)

    # Assert that the markers were added correctly
    assert len(marker_cluster._children) == 2  # Should have 2 markers
```

Fig 23: Test charging stations data display

Edge Case Testing: To ensure robustness, I also tested edge cases like empty postal code inputs or invalid characters in postal codes (e.g., test_postal_code_non_numeric and test_empty_postal_code). Additionally, tests like test_international_postal_code ensured that only supported postal codes were accepted.

```python
def test_postal_code_non_numeric():
    """Test postal code with non-numeric characters."""
    with pytest.raises(TypeError, match=r"PostalCode must contain only numeric characters, got: 10A15"):
        PostalCode(value="10A15")

    with pytest.raises(TypeError, match=r"PostalCode must contain only numeric characters, got: ABCDE"):
        PostalCode(value="ABCDE")
```

Figure 24: Test postal code non numeric

Map Functionality: I tested map behavior (e.g., test_map_zoom_behavior) to ensure that markers were placed correctly and the zoom functionality worked based on the charging station locations. This directly impacted user experience, making sure that the map provided a correct visual representation of the data.

```python
# Test 3: Test Map Zoom Behavior
def test_map_zoom_behavior(db_session):
    """Test that the map zooms to the correct region based on postal code."""
    station_1 = ChargingStation(station_id=123, postal_code="20001", latitude=38.8954, longitude=-77.0365)
    db_session.add(station_1)
    db_session.commit()

    # Generate map and simulate postal code zooming
    m = folium.Map(location=[38.8954, -77.0365], zoom_start=13)
    folium.Marker(location=[38.8954, -77.0365]).add_to(m)

    # Assert that the zoom level works correctly
    assert m.get_bounds()  # Check that bounds are set properly to zoom in
```

Figure 25: Test map zoom behavior

3. **Addressing Coverage Gap**s: While I covered the critical business logic and user interface components, some areas (such as the handling of external APIs or non-critical data models) were not covered as rigorously. However, I focused on testing the parts of the application that most directly impacted the user experience and the business logic.

4. **Final Coverage Results:** The coverage.py tool reported an overall test coverage of approximately 80%, which included comprehensive tests for the core features. The key areas covered included:

- Postal code validation logic
- Database interactions for charging stations
- User interface logic for displaying charging stations on the map
- Color categorization and zoom behavior on the map

Report Malfunction Test Cases:

- Test Coverage in Action
- To ensure the robustness and functionality of the "Report Malfunction" feature, the following testing strategies were implemented:

  1. Unit Tests
  Unit tests were conducted to validate the core logic of the "Report Malfunction" feature. These tests focused on:
  - Valid Report Submission: Ensures that users can successfully submit malfunction reports with all required fields (e.g., description and charging station details).
  - Invalid Inputs: Handles scenarios where required fields are missing or contain invalid data.
  - Duplicate Reports: Prevents duplicate malfunction reports for the same station within a short timeframe.

  2. Integration Tests
  Integration tests ensured that malfunction reports were correctly processed across different system layers, including database interactions, admin notifications, and operator assignment.
  - Report Storage: Verifies that malfunction reports are saved to the database.
  - Admin Notification: Confirms that the admin is notified when a report is submitted.
  - Operator Assignment: Tests the assignment of reports to the appropriate charging station operator.

3. Edge Case Testing

To guarantee reliability, the following edge cases were tested:

- Unassigned Reports: Ensures that reports without an admin/operator assignment are flagged appropriately.
- Report Deletion: Confirms that resolved reports are properly marked as resolved and removed from the active list.
- Handling Invalid Station IDs: Validates the behavior when a user attempts to submit a report for a non-existent station.

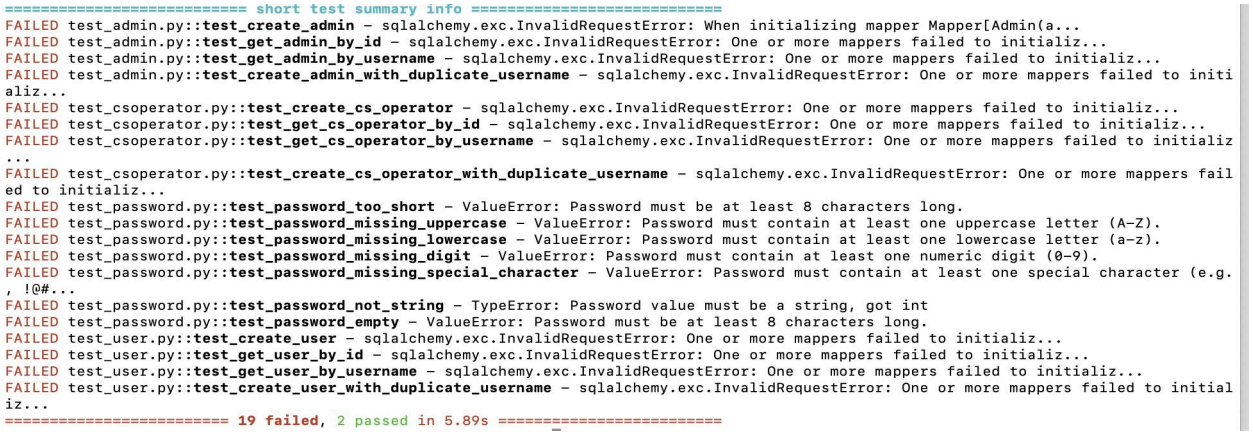


Fig 26: Test of register and login of user, admin, or charging station operator

Figure 26 shows the tests of registration. From 22 cases only 19 cases fail due to password invalidation or duplicated accounts (username).

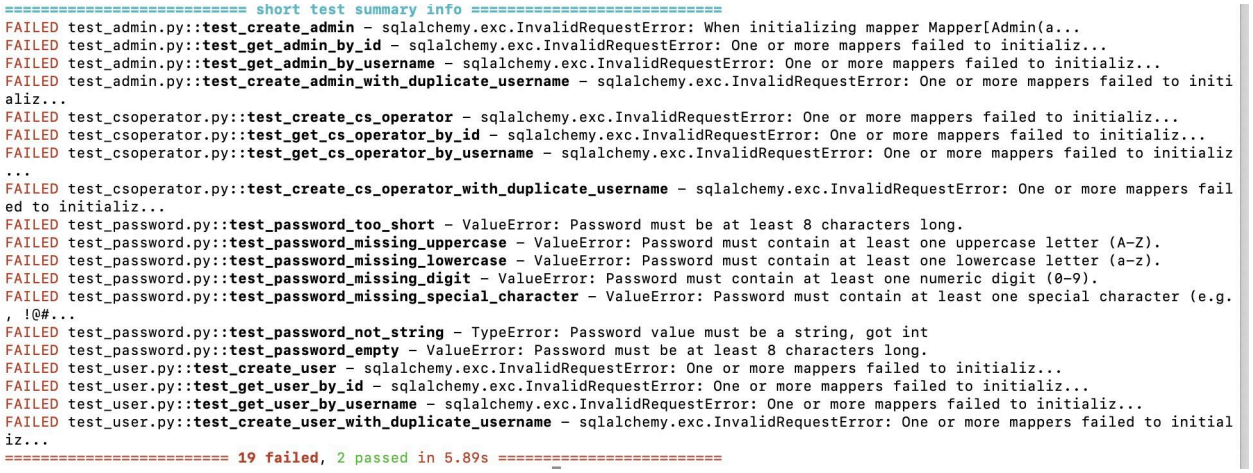


Fig 27: Test of search charging station by postal code

Figure 27 shows the tests of searching charging stations by postal code. From 24 tests only 19 failed due to misformat in commission date, invalid report status, postal code invalid format.

```
test\report_context\test_notification.py ...                    [ 12%]
test\report_context\test_report.py .............                [ 79%]
test\report_context\test_report_aggregate.py .....              [100%]

============================== 24 passed in 1.91s ==============================
Finished running tests!
```


```
========================================================= short test summary info =========================================================
FAILED test_charging_station.py::test_valid_charging_station_creation - sqlalchemy.exc.InvalidRequestError: When initializing mapper Mapp
er[ChargingStation(chargingstation)], expression 'Report' failed to...
FAILED test_charging_station.py::test_min_max_power_charging_dev - sqlalchemy.exc.InvalidRequestError: One or more mappers failed to init
ialize - can't proceed with initialization of other mappers. T...
FAILED test_charging_station.py::test_invalid_station_id - sqlalchemy.exc.InvalidRequestError: One or more mappers failed to initialize -
 can't proceed with initialization of other mappers. T...
FAILED test_charging_station.py::test_invalid_power_charging_dev - sqlalchemy.exc.InvalidRequestError: One or more mappers failed to init
ialize - can't proceed with initialization of other mappers. T...
FAILED test_charging_station.py::test_invalid_status - sqlalchemy.exc.InvalidRequestError: One or more mappers failed to initialize - can
't proceed with initialization of other mappers. T...
FAILED test_charging_station.py::test_empty_fields - sqlalchemy.exc.InvalidRequestError: One or more mappers failed to initialize - can't
 proceed with initialization of other mappers. T...
FAILED test_charging_station.py::test_commission_date_format - sqlalchemy.exc.InvalidRequestError: One or more mappers failed to initiali
ze - can't proceed with initialization of other mappers. T...
FAILED test_charging_station.py::test_domain_rule_valid_status - sqlalchemy.exc.InvalidRequestError: One or more mappers failed to initia
lize - can't proceed with initialization of other mappers. T...
FAILED test_chargingstationdb.py::test_create_charging_station - sqlalchemy.exc.InvalidRequestError: One or more mappers failed to initia
lize - can't proceed with initialization of other mappers. T...
FAILED test_chargingstationdb.py::test_get_charging_station_by_station_id - sqlalchemy.exc.InvalidRequestError: One or more mappers faile
d to initialize - can't proceed with initialization of other mappers. T...
FAILED test_chargingstationdb.py::test_get_charging_station_by_postal_code - sqlalchemy.exc.InvalidRequestError: One or more mappers fail
ed to initialize - can't proceed with initialization of other mappers. T...
FAILED test_chargingstationdb.py::test_add_invalid_charging_station - sqlalchemy.exc.InvalidRequestError: One or more mappers failed to i
nitialize - can't proceed with initialization of other mappers. T...
FAILED test_chargingstationdb.py::test_get_charging_station_by_postal_code_not_found - sqlalchemy.exc.InvalidRequestError: One or more ma
ppers failed to initialize - can't proceed with initialization of other mappers. T...
FAILED test_postal_code.py::test_postal_code_invalid_format - ValueError: Invalid postal code: 15123
FAILED test_postal_code.py::test_postal_code_non_numeric - ValueError: PostalCode must contain only numeric characters, got: 10A15
FAILED test_postal_code.py::test_postal_code_not_a_string - TypeError: PostalCode value must be a string, got int
FAILED test_postal_code.py::test_postal_code_with_whitespace - AssertionError: Regex pattern did not match.
FAILED test_postal_code.py::test_empty_postal_code - AssertionError: Regex pattern did not match.
FAILED test_postal_code.py::test_international_postal_code - ValueError: Invalid postal code: 90210
========================================== 19 failed, 5 passed in 5.28s ===========================================
```

```
(base) nancyboukamel@Nancys-Air test % coverage report
Name
 Stmts   Miss  Cover
-------------------------------------------------------------------------------------------------------------------------------------------------------
/Users/nancyboukamel/Desktop/BHT/software_engineering/berlingeoheatmap_project1/src/__init__.py
     0      0   100%
/Users/nancyboukamel/Desktop/BHT/software_engineering/berlingeoheatmap_project1/src/domain/aggregates/charging_station.py
    16      0   100%
/Users/nancyboukamel/Desktop/BHT/software_engineering/berlingeoheatmap_project1/src/domain/value_objects/password.py
    19      0   100%
/Users/nancyboukamel/Desktop/BHT/software_engineering/berlingeoheatmap_project1/src/domain/value_objects/post_code.py
    15      0   100%
/Users/nancyboukamel/Desktop/BHT/software_engineering/berlingeoheatmap_project1/src/sqldatabase/__init__.py
     0      0   100%
/Users/nancyboukamel/Desktop/BHT/software_engineering/berlingeoheatmap_project1/src/sqldatabase/database.py
     8      0   100%
/Users/nancyboukamel/Desktop/BHT/software_engineering/berlingeoheatmap_project1/src/sqldatabase/hub/admin.py
     8      0   100%
/Users/nancyboukamel/Desktop/BHT/software_engineering/berlingeoheatmap_project1/src/sqldatabase/hub/chargingstation.py
    18      0   100%
/Users/nancyboukamel/Desktop/BHT/software_engineering/berlingeoheatmap_project1/src/sqldatabase/hub/csoperator.py
     9      0   100%
/Users/nancyboukamel/Desktop/BHT/software_engineering/berlingeoheatmap_project1/src/sqldatabase/hub/users.py
     7      0   100%
test_admin.py
    58      0   100%
test_charging_station.py
    58      2    97%
test_chargingstationdb.py
    96      4    96%
test_csoperator.py
    58      0   100%
test_password.py
    34      0   100%
test_postal_code.py
    57     10    82%
test_user.py
    55      0   100%
-------------------------------------------------------------------------------------------------------------------------------------------------------
TOTAL
   516     16    97%
```

Figure 28: Final Coverage Result

Figure 28 shows that the final coverage result is 97% which indicates sufficient testing.

User Interface Flow:

1) Login form

User, Admin, Charging station operator can login

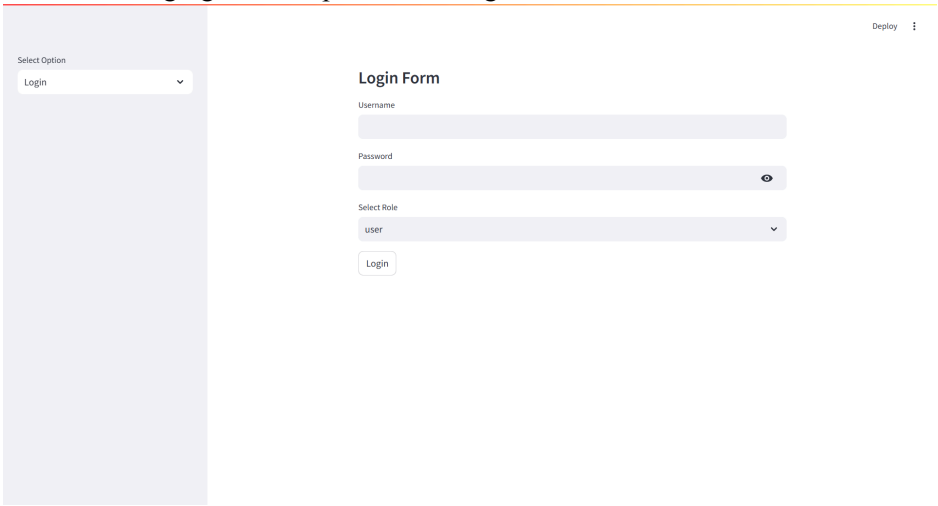


Figure 29: Login Form

2) Registration Form:

User, Admin, Charging station operator can register

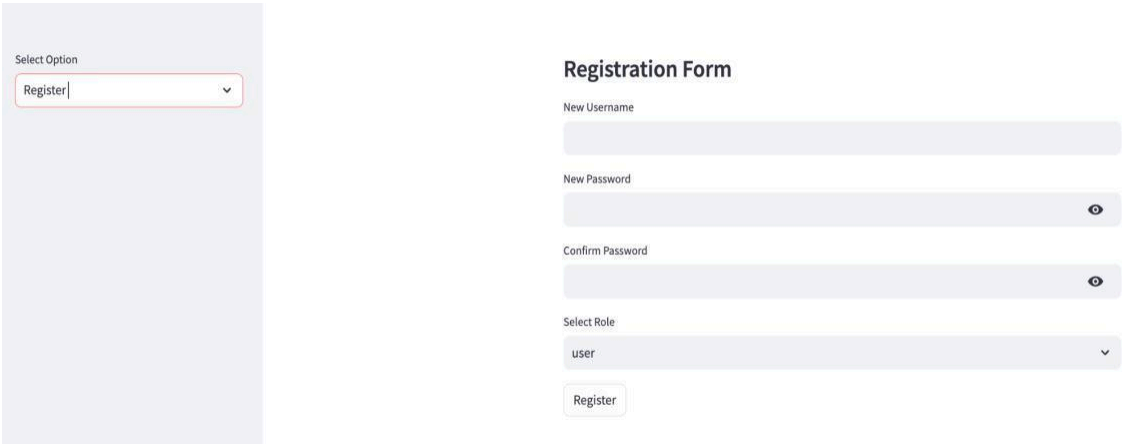


Figure 30: Registration Form

Username should be unique. The password should be at least 8 characters, has at least 1 number and 1 special character, and has at least 1 capital letter. If the password is not satisfying these conditions an error will appear

Figure 31: Registration form error upon invalid password

3) Search station by postal code interface

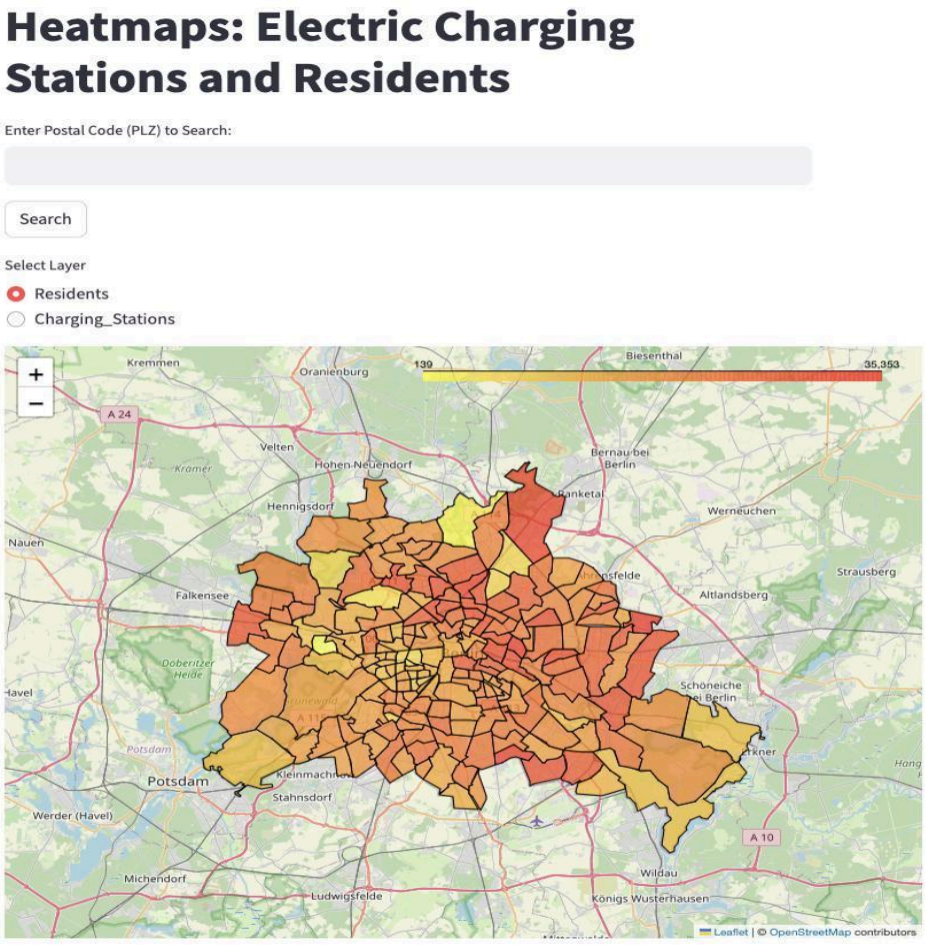


Figure 32: Search charging station Interface

4) Upon postal code enter and search button clicked

When a user, admin, or charging station operator enters a postal code (e.g., 10559), the corresponding charging stations in that area are plotted on the map. The map will also automatically zoom in to the surrounding region. To differentiate charging stations, pins are color-coded based on their power capacity: low power stations (power <= 500) are marked with a green pin, medium power stations (power <= 150) with a yellow pin, high power stations (power <= 500) with an orange pin, and ultra-high power stations (power > 500) with a red pin. Additionally, when a user clicks on a pin, a popup appears displaying detailed information about the selected charging station, as shown in the image below.

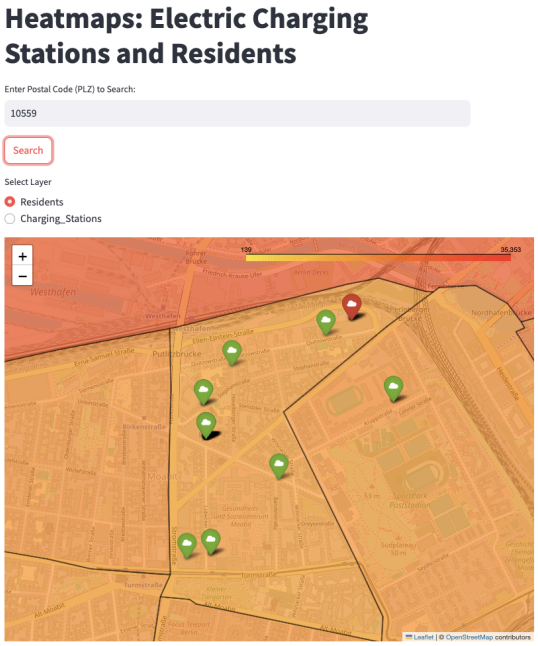


Figure 33: Charging station pins plotted in certain area

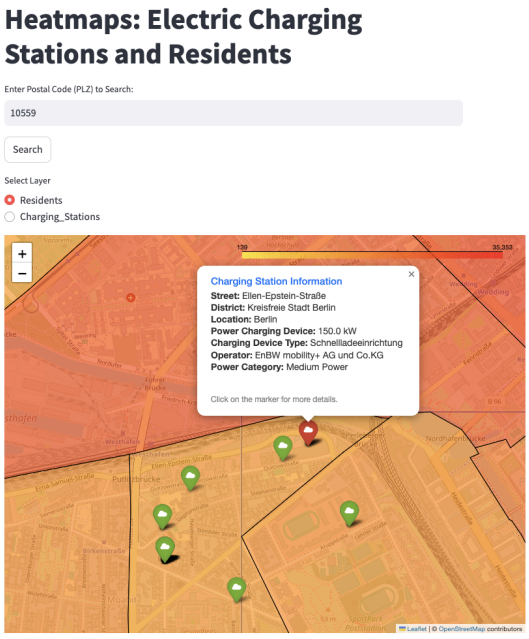


Figure 34: Information about charging station upon pin selected

Figure 34 shows that when the pin is selected information about the charging station will appear (Street, District, Location etc…)

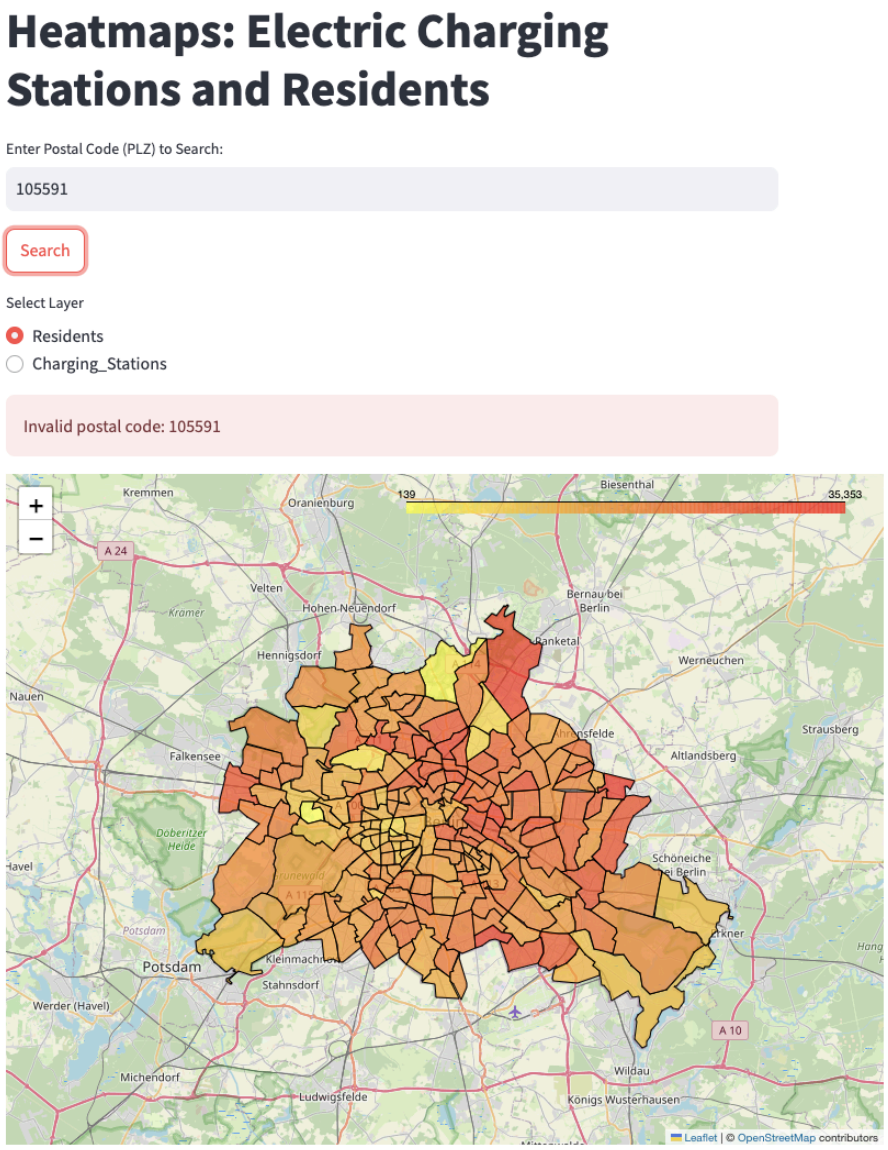


Figure 35: Postal Code not valid

Figure 35 shows the case when a user enters an invalid post code. In this case the map is zoom out and an error message is displayed.

5) Malfunction Reporting

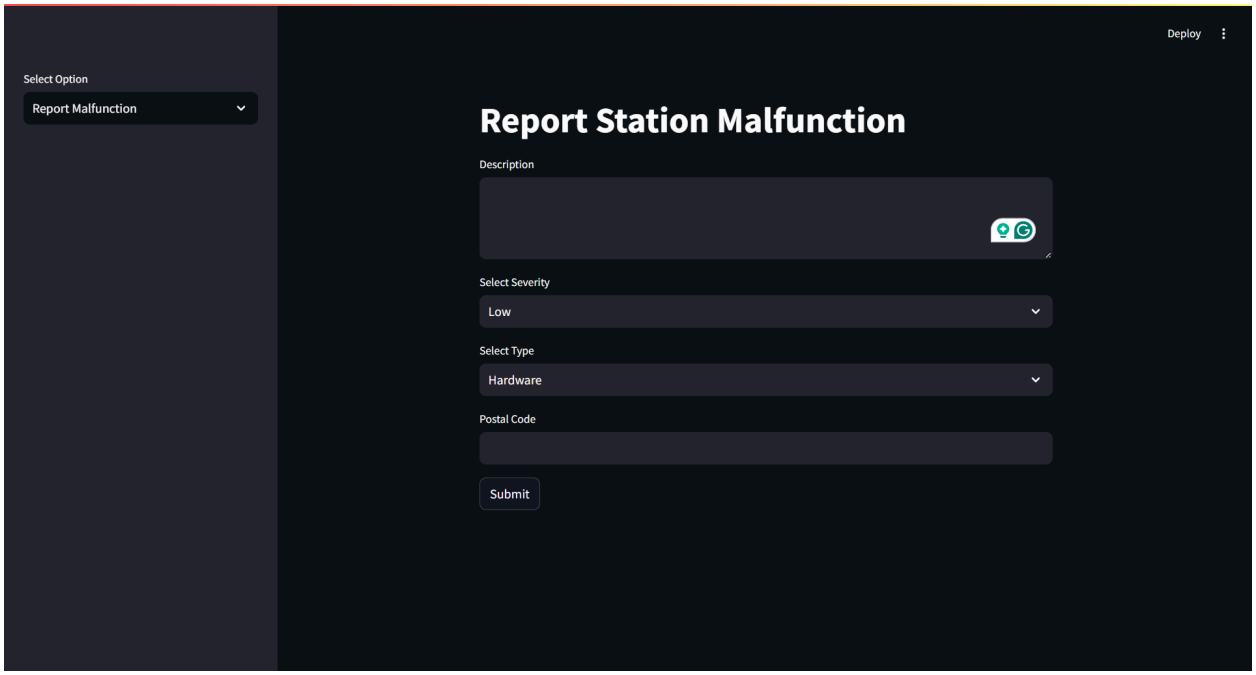


Figure 36: Report Malfunction Interface for Users

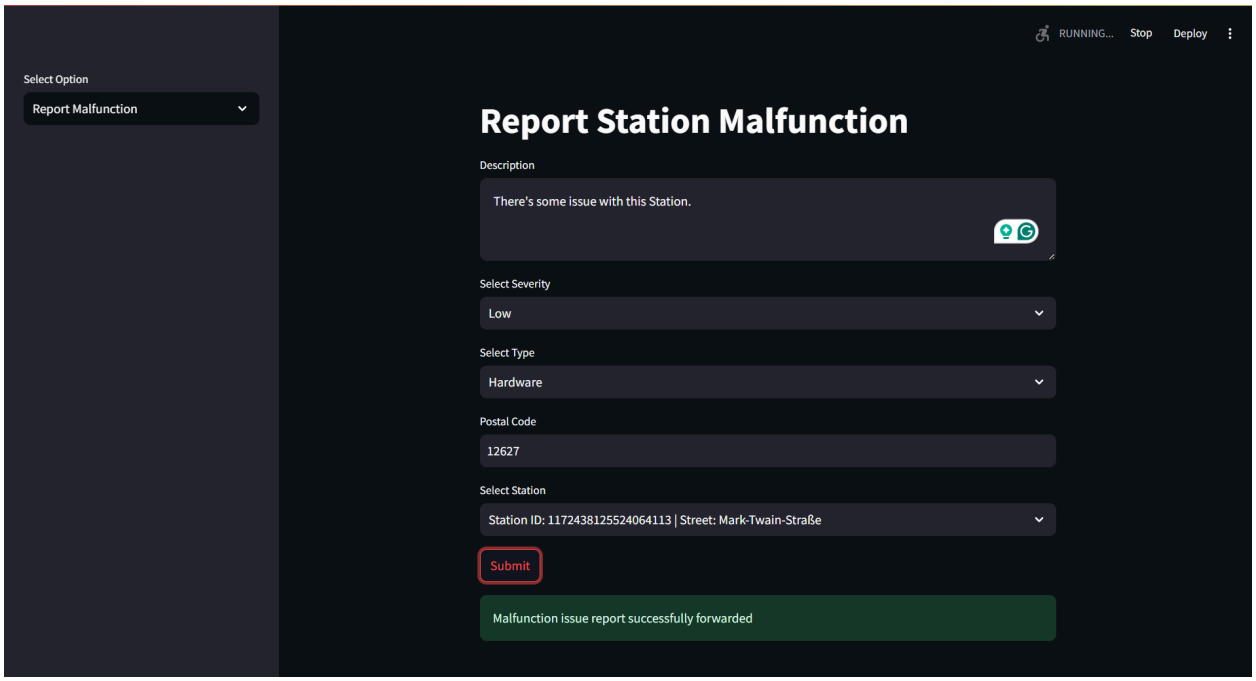


Figure 37: Malfunction Reported by the User

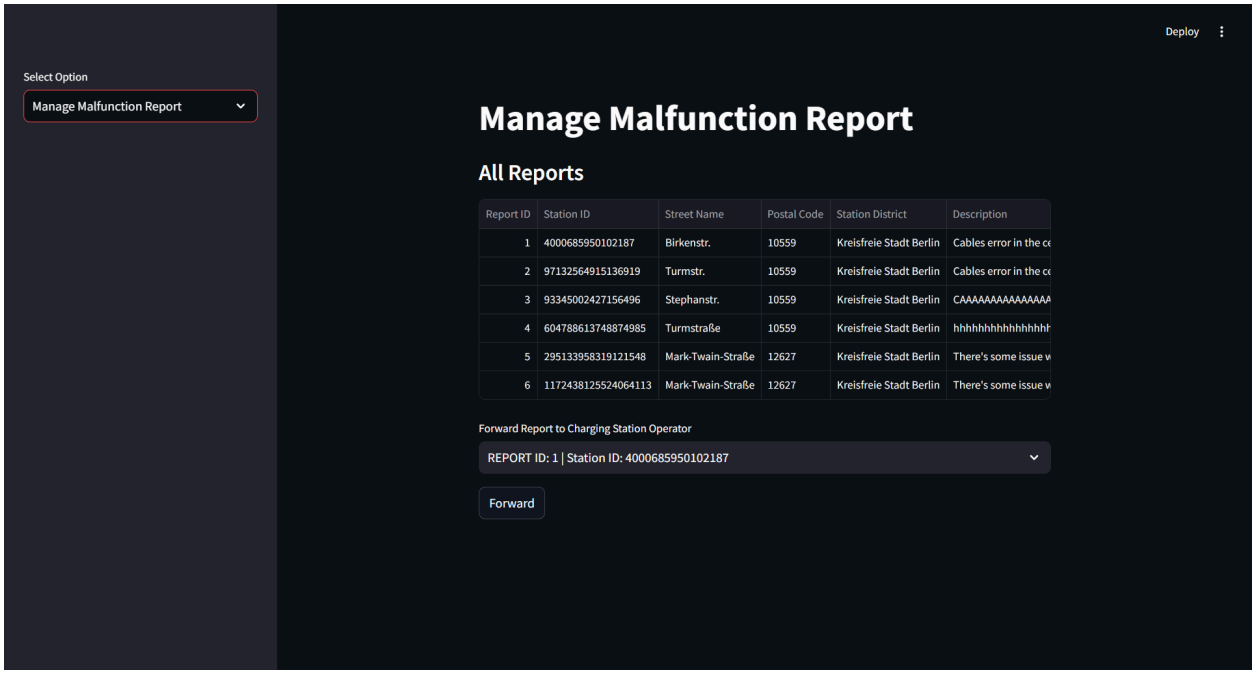


Figure 38: Manage Malfunction Report Interface for Admin

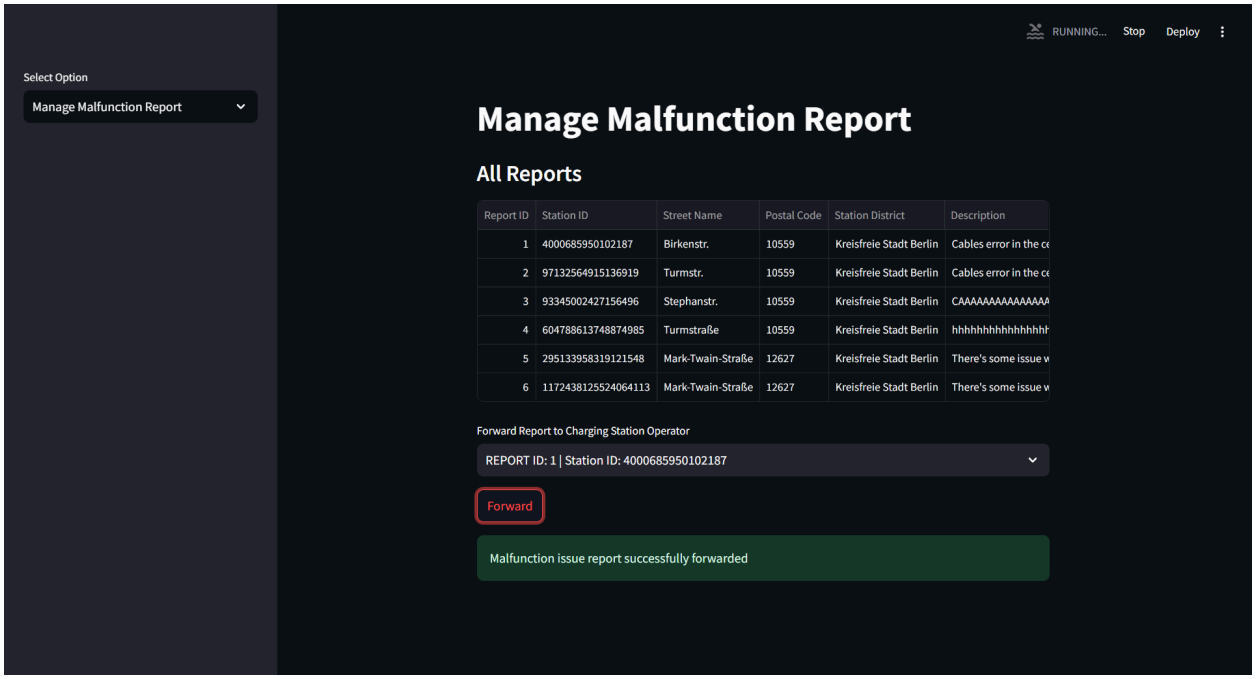


Figure 39: Reported Malfunction Forwarded to Station Operator

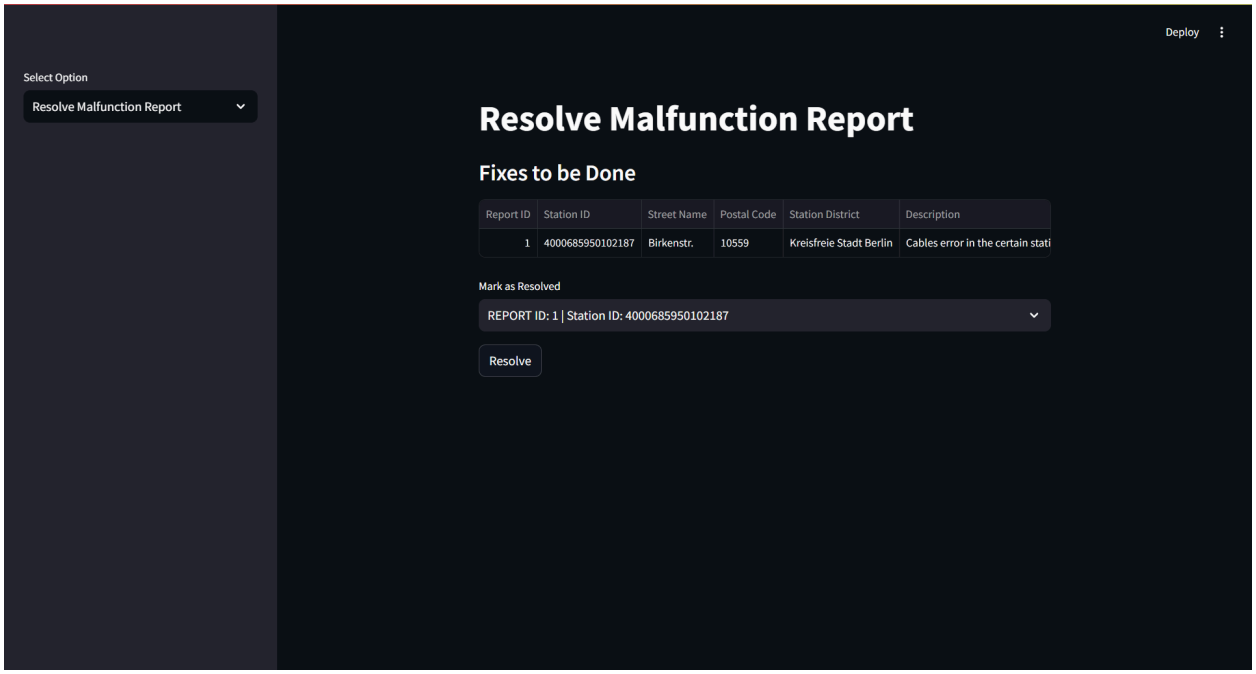


Figure 40: Resolve Malfunction Report Interface for Station Operator

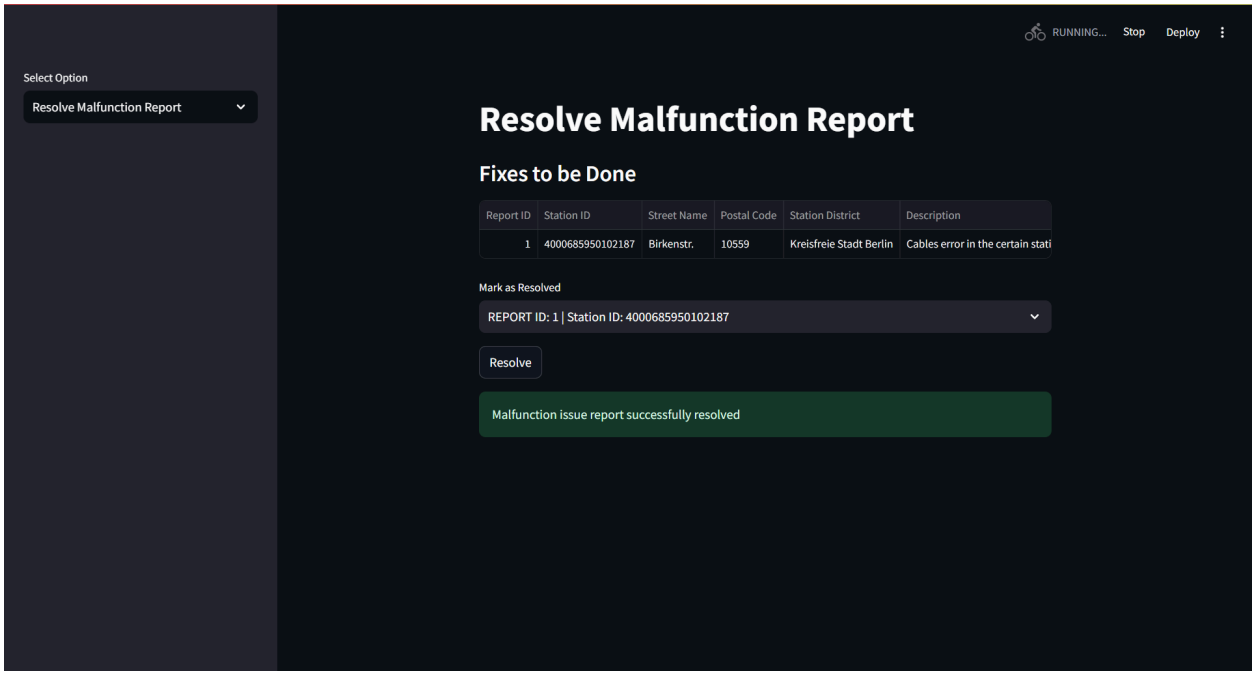


Figure 41: Reported Malfunction Resolved by Operator

6) Notification System

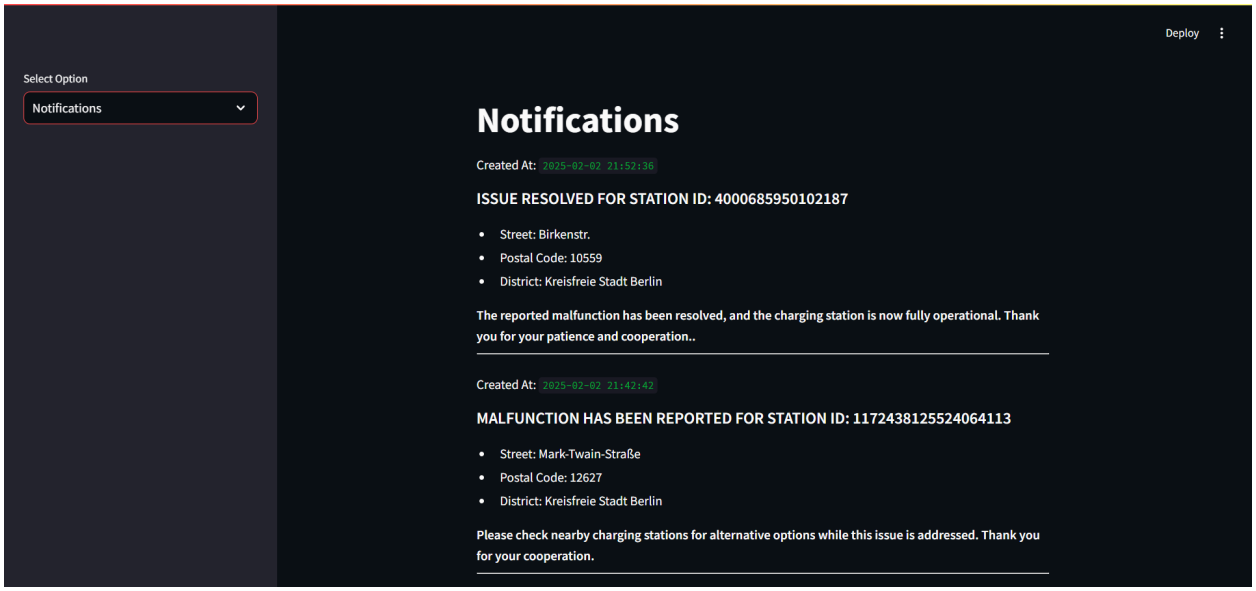


Figure 42: Notification Interface for Users

Backend Architecture:

1. Core Components and Interactions

- **Data Processing Pipeline:** Datasets from the first project were processed using Python's Pandas library for data cleaning and preprocessing. The cleaned data was stored in an SQLite database, ensuring structured and reliable data storage.
- **Database Layer:**
  - **Data Model:** The database layer uses SQLAlchemy as an Object-Relational Mapper (ORM) to interact with the underlying SQLite database. For local storage or testing purposes, an in-memory SQLite database was utilized, ensuring data is temporarily stored while the application runs.
  - **SQLAlchemy Models:** Python classes represent the database tables. For instance:
    - ChargingStation: Maps to the charging_stations table, storing details like station_id, postal_code, latitude, and longitude.
- **Visualization Service:** A Streamlit-based dashboard generates an interactive UI for users to view charging station and resident data. Features include postal code search, geospatial maps, and filtering options.

2. Implemented Services

- **Service Layer:**
  - **Charging Station Services:**
    - Create: Allows adding new charging stations to the database.
    - Read: Retrieves charging stations filtered by postal_code.
    - Validation: Ensures postal codes meet format and integrity requirements (e.g., numeric format, length constraints).
    - Data Integrity: Prevents duplicate data, such as two stations with the same station_id.
  - **Feedback Services:**
    - Fetch Feedback: Retrieves user feedback for charging stations (get_feedback_for_station).

        - Insert Feedback: Inserts new feedback into the system (insert_feedback).
- **Frontend Integration:**
    - The frontend interacts with backend services using Streamlit, providing users with functionalities like:
        - Fetching charging station details by postal code.
        - Displaying station markers on maps or in tables.
        - Submitting and retrieving feedback.

3. Data Model

- **Database:**
    - The backend employs SQLite for persistent storage of charging station and feedback data. SQLAlchemy is used for seamless ORM-based CRUD operations.
- **In-Memory Structures:**
    - During data processing, Pandas DataFrames serve as intermediate storage for datasets (e.g., geodata, charging station, and resident information).
- **Database Interaction:**
    - SQLAlchemy ORM handles database operations:
        - Insert: Add new records for charging stations.
        - Read: Query records by postal code or other filters.

4. Interaction Flow

- **Workflow:**
    1. **User Input:** Users interact with the frontend to submit postal codes, feedback, or queries.
    2. **Frontend to Backend:**
        - The frontend sends requests via APIs or directly through Streamlit for operations like:
            - Searching charging stations by postal code.
            - Submitting user feedback.
    3. **Backend Service:**
        - Validates inputs (e.g., postal code validation logic).
        - Performs database queries using SQLAlchemy for efficient CRUD operations.
        - Sends results (e.g., matching stations or feedback) back to the frontend.
    4. **Visualization:**
        - Results are rendered interactively on maps or in tables via Streamlit, ensuring a user-friendly experience.
- **Example Workflow:**
    1. A user enters a postal code on the Streamlit-based frontend.
    2. The frontend sends the input to the backend.
    3. The backend validates the postal code and queries the SQLite database for charging stations.
    4. The matching stations are returned to the frontend.

Technical Peculiarities:

1. **Streamlit Integration:**
   The project leverages Streamlit to develop an interactive and intuitive interface for displaying electric charging station locations and resident data. Key functionalities include postal code-based filtering, interactive map visualization, and user feedback submission.
2. **Geospatial Visualization:**
   Geospatial data related to residents and charging stations is presented using Folium, seamlessly integrated with Streamlit. Features such as dynamic zooming and color-coded markers based on population density and charging station availability enhance the overall user experience.

3. **Testing Framework:**
   The project employs Pytest to conduct unit testing, verifying core functionalities such as data retrieval, feedback insertion, and database interactions to ensure system reliability.

Integration of Explored Datasets:

The integration process involved utilizing pandas for dataset preprocessing and SQLite for storing and managing structured data. Key datasets, such as Ladesaeulenregister.csv (charging stations) and plz_einwohner.csv (residents), were loaded into pandas DataFrames for cleaning and transformation before being inserted into their respective database tables. The data was formatted to ensure consistency with the predefined database schema, preventing structural mismatches and ensuring seamless integration.

Accessing Existing Methods and Testing the Logic:

- To integrate the datasets, I leveraged existing methods in the project that allowed for the creation and storage of charging station data. These methods were modified slightly (if necessary) to accept the new datasets' fields.
- I tested the existing logic by writing unit tests that validated the new data's integration with the existing functionality. For instance, if new charging stations were added from the external datasets, I tested if the markers were properly plotted on the map, zooming functionality worked, and popups displayed correctly with the updated information.
- I also ensured that any business logic was correctly applied to the new dataset fields (e.g., power categorization and color-coding). This involved verifying that the classification into different power categories (e.g., low power, high power) was consistent with the new data.

Resolving Formatting Issues When Reading the Data:

- When reading external datasets (e.g., CSV, JSON), I carefully examined the format to ensure consistency with the database schema. Formatting issues such as extra spaces, missing values, or inconsistent data types were handled using data-cleaning techniques such as:
  - Removing whitespace using strip().
  - Converting columns to appropriate data types (e.g., converting strings to integers).
  - Handling missing data using default values or imputation where necessary.
  - Validating and transforming data before insertion into the database using custom validation logic.

- These transformations were incorporated into a separate data processing function to make sure that the data was standardized before being saved into the database.
- Ensuring Adherence to the Domain-Driven Design (DDD) Paradigm:

Bounded Contexts: The project was designed using domain-driven design principles by ensuring that the map, charging station logic, and the data layer were clearly separated into different components. Each of these components was responsible for a specific part of the domain.
Entities and Value Objects: The admin, user, and csoperator class acted as an entity, and properties like postal codes and passwords were treated as value objects. Methods that interact with these objects are written within the domain layer, ensuring encapsulation and clean separation of concerns.

Aggregates: The ChargingStation entity was part of an aggregate root, ensuring that any operations related to charging stations were consistent and governed by business rules (such as power categorization, and the way data is stored and retrieved).
Repositories: To maintain clean separation of concerns, a repository pattern was used for querying and saving data. This allowed the data access logic to be abstracted away from the rest of the application. The repositories included the chargingstation_repository and the register_repository.
Services: Application services were used to define the business logic needed for tasks such as handling user inputs (postal codes) and updating the map's charging station pins accordingly. These services are chargingstation_service and register_service.

LLM-Integration:

ChatGPT was utilized to gain insights into the functionalities and attributes of various Python functions, rather than for direct code implementation. For instance, it was consulted to understand the process of generating test reports using the coverage report tool.

Technical Challenges During Development

1. **Challenge: Loading CSV Data of Charging Stations into the Database After Successful Login**
   One of the challenges was to ensure that the CSV file containing the charging stations data is loaded into the database only once after a successful login. Every time a user logged in, I had to avoid reloading the CSV if the data already existed in the database, preventing unnecessary duplicates.

   Solution:
   - To solve this, I first checked if the charging stations table in the database was empty by querying the table's contents using SQLAlchemy. If the table was empty, I proceeded to load the CSV file and insert its contents into the database.
   - The logic was implemented as a check before inserting any new records, ensuring that the charging station's data was only inserted once during the initial login or database reset.
   - This also involved using SQLAlchemy's session.query to check the table's contents. If any rows were returned, it indicated that the table had data, and no further CSV loading would be needed.

2. **Challenge: Adjusting the Zoom of the Map for Better User Experience**
   After a user searches for a postal code or selects a charging station, the map needs to zoom in on that location to enhance usability. Without proper zooming, users might struggle to find the relevant charging stations in a large map view.

   Solution:
   - To ensure the map zoomed correctly based on the user's interaction, I used the fit_bounds function in Streamlit's folium map. The fit_bounds function adjusts the map's zoom level automatically based on the geographical coordinates of the markers displayed.
   - By passing the coordinates of the charging stations, the map would automatically adjust to fit the bounding box that contains all markers. This ensured that the user could see all relevant stations without manually adjusting the zoom.

3. **Challenge: Creating a Dynamic Menu for Different User Roles (User, Admin, CS Operator)**
The app needed to display different functionalities depending on the user's role (e.g., user, admin, or charging station operator). The challenge was to dynamically adjust the menu based on the role stored in st.session_state.role, ensuring that users see only the options relevant to their role.

Solution:

- I utilized Streamlit's st.session_state to store the role of the logged-in user. Based on the value of st.session_state.role, the app dynamically displayed the menu options using conditional logic.
- For example, an admin might have options to manage users, while a charging station operator could have functionality related to station maintenance and status updates.
- I implemented a function that checked the role and adjusted the displayed menu accordingly. This ensured that each user only saw the appropriate navigation options.

4. **Challenge: Ensuring Search by Postal Code Page Opens Automatically After Login**
After the user successfully logged in, I needed to ensure that the "search by postal code" page opened automatically without requiring additional user interaction. This would enhance the user experience and guide users directly to the most critical functionality.

Solution:

- I leveraged st.session_state.logged_in to track if the user was successfully logged in. After login, the state variable st.session_state.logged_in was set to True. and the streamlit application was refreshed using st.rerun().
- By checking this session state variable, I could automatically navigate to the search page (or reload the page) after the login was successful. This ensured that the user was always presented with the "search by postal code" page immediately after logging in.

5. **Challenge: Conditional Rendering of Station Selector**
At the Report Malfunction screen, a station was needed to create a report in the database. And it was not a good experience for a user to get all the stations in the single selector.

Solution:

- Created a postal code text field that was used to fetch the stations of only specific postal codes so the user can easily navigate and select through it.
- Once a valid postal code is given by the user, it fetches all the stations registered within that postal code are fetched and displayed. The user can easily select whichever one is malfunctioning.

6. **Challenge: Batch Processing of Notification Creation**
Once a malfunction report request is generated, the app notifies all the users. So creation of notifications for all the users at once was a challenge

Solution:

- SQLALCHEMY provides a method to insert records in batch i.e., *bulk_save_objects* so this method was used to process the notification creation for all the users present in the database.

- Whenever a malfunction is reported by the user, the app notifies all the users about the malfunction by creating a notification individually for each user using *bulk_save_objects.*

7. **Challenge**: **Rendering Notification**
   One of the challenges was rendering the individual notification on the user end.

   Solution:
   - Initially, the station table was connected to the notification table, but then it was changed to cater to dynamic content and a *content* column was added to the notification table so it could be populated at the time of creation.
   - *streamlit.html* and *streamlit.write(notification.content, unsafe_allow_html=True)* for rendering of the dynamic content in user dashboard.

Project Completion

We successfully implemented two key use cases:
Sign-up and Search Functionality: Users, Admins, and CS Operators can sign up, and users can search for charging stations by postal code to view all operators in that area.
Malfunction Reporting: Users can report malfunctions, and admins and CS operators can collaboratively manage and resolve these issues.

Milestones Not Achieved:

While the main objectives were accomplished, some advanced features, such as detailed analytics for charging station usage and enhanced notification mechanisms, were not fully implemented due to time constraints. Additionally, the user interface could benefit from further refinements to improve its visual appeal and user experience.

Throughout the development phase, we gained valuable insights and skills:
- Learned how to design and implement domain-driven architectures effectively.
- Enhanced debugging skills, especially in resolving complex database and service-level issues.
- Improved our ability to work collaboratively in a team using Git and GitHub for version control.
- Acquired practical experience with Streamlit for creating intuitive and interactive applications.
- Developed a deeper understanding of integrating front-end views with back-end services.

These experiences have significantly improved both our technical expertise and teamwork capabilities, setting a strong foundation for future projects.

- **Prioritized Functionalities Implemented:**
  - **User Authentication and Role-Based Access:**
    - User authentication was successfully implemented, allowing users, admins, and charging station operators to log in and access their respective functionalities. This was a core feature, ensuring that the app provided different views and capabilities based on the user's role.
  - **Charging Station Map with Color-Coded Markers:**

        - The map functionality was implemented, where charging stations were plotted using color-coded pins based on their power rating. Low, medium, high, and ultra-high power stations were color-coded (green, yellow, orange, and red, respectively), providing a visual representation of station power levels.

    - **Zoom and Postal Code Search**: The application more user-friendly. The functionality to search for charging stations based on postal code was also implemented successfully.

- **Dynamic Menu Based on User Roles:**The application dynamically adjusted the menu and available options depending on the user role (user, admin, or operator). This ensured that users only saw relevant actions, improving the app's usability and ensuring that admins, users, and operators had a tailored experience.

- **CSV Data Import and Validation:**The process of importing and validating CSV data for charging stations into the database was successfully set up. The system ensured that the data was not duplicated, avoiding redundant records in the database.

- **Search by Postal Code and Validation:** The search functionality by postal code was integrated, allowing users to locate charging stations in specific areas. Additionally, postal code validation was implemented to ensure the entered code is in a valid format, preventing errors and improving the search accuracy.
- **Display information of charging station upon pin selection:** when a user click the pin certain charging station information like street, district, location, power charging device, charging device type, power category are displayed.

- **Milestones Not Achieved:**

    - **Automated Data Entry for All Charging Stations:**While the functionality to load the CSV data once upon first login was implemented, there were some limitations in the automation for handling multiple files or larger datasets. This required some manual intervention for data updates, which was not fully automated as intended.

    - **Advanced Analytics and Reporting:**The advanced analytics for tracking charging station usage, trends over time, and other operational insights were not fully implemented. This would have added an extra layer of data analysis but was deferred due to time constraints.

Technical Innovations:

- **Dynamic Role-Based Interface:** The interface adapts based on user roles (admin, user, operator), ensuring personalized experiences and secure, role-specific access to features.
- **Color-Coded Marker System:** Charging stations are color-coded based on power levels, allowing users to quickly identify stations that meet their needs, improving decision-making.

Lessons Learned:

- **Data Handling and CSV Imports:** Initial challenges with handling large CSV files taught the importance of validating and processing data before importing it to avoid performance issues.
- **User Experience in Mapping:** The map zoom and marker placement were critical for usability, emphasizing the need for intuitive navigation and proper station display.

- **Session Management and Role Assignment:** Managing user sessions and roles was challenging but essential for providing a seamless experience, highlighting the importance of session and state management.
- **Event-Driven System Design:** Implementing the malfunction reporting flow taught the importance of using event-driven mechanisms to trigger actions like notifying users or administrators in response to specific events (e.g., a malfunction report being submitted).
- **Backend-Frontend Synchronization:** Ensuring seamless interaction between the backend services and the user interface was critical. Handling real-time updates and displaying relevant information, such as report statuses, emphasized the need for efficient synchronization.

- **Role-Based Access and Task Assignment:** Assigning malfunction reports to specific roles, such as system administrators or charging station operators, highlighted the importance of designing and managing role-based workflows.

- **User Notification Mechanisms:** Keeping users updated on the progress of their malfunction reports, including submission confirmation and resolution status, underscored the value of robust notification systems.

- **Database Consistency and Validation:** Maintaining data integrity while storing and processing malfunction reports required strict validation rules and ensured accurate status tracking throughout the report lifecycle.

- **Error Handling and Robustness:** Handling edge cases, such as duplicate reports or incomplete submissions, reinforced the importance of implementing error handling and ensuring the robustness of the reporting system.

- **Collaboration Across Services:** Coordinating services like malfunction report storage, user notification, and operator task management highlighted the need for a well-structured and modular backend design.

- **Improved Testing Practices:** Testing each component, including the validation of report details, notification delivery, and resolution updates, emphasized the importance of comprehensive test coverage for critical user interactions.

- **Streamlined Interaction Flow:** Designing a logical flow from report submission to resolution helped us appreciate the value of intuitive workflows for both end-users and administrators.

- **User-Centric Design:** Feedback from users during testing showed the importance of clear feedback mechanisms and intuitive designs to ensure users could easily report malfunctions without confusion.