Design and Implementation of a Django-Based Webpage for Real-Time Receiver Status Monitoring Using PostgreSQL

INTERNSHIP REPORT

Submitted in partial fulfillment of

the requirement for the award of an M.Tech Degree in

Computer Science and Engineering with specialization in

Computer Science and Engineering

of the APJ Abdul Kalam Technological University

Submitted By

ARYA VARGHESE

Register No : TVE23CSCE05

Third Semester

M.Tech Computer Science and Engineering

with specialization in Computer Science and Engineering

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COLLEGE OF ENGINEERING TRIVANDRUM

AUGUST 2024

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COLLEGE OF ENGINEERING TRIVANDRUM

CERTIFICATE

This is to certify that this internship report entitled **Design and Implementation of a Django-Based Webpage for Real-Time Receiver Status Monitoring Using PostgreSQL** *is a bonafide record of internship done by* **ARYA VARGHESE** *under our guidance towards the partial fulfillment of the requirements for the award of the Degree of* **Master of Technology in Computer Science and Engineering,** of the **APJ Abdul Kalam Technological University** during the year 2023-2025.

Dr. Salim A	E. Soorej Kumar
Professor	COO, AugsenseLab Pvt Ltd
Head of the Department	

ACKNOWLEDGEMENT

It is with great enthusiasm and spirit I am bringing out this internship report. I also feel that it is the right opportunity to acknowledge the support and guidance that came in from various quarters during the course of the completion of my internship.

The project **Design and Implementation of a Django-Based Webpage for Real-Time Receiver Status Monitoring Using PostgreSQL** was a great learning experience for me .I am submitting this work to AugsenseLab Pvt Ltd, TrestPark.

I am glad to mention **E. Soorej Kumar** for his guidance on this project. His guidance and support helped me gain perspective and overcome various obstacles.

I am grateful to my mentor **Dr Divya S Vidyadharan** for motivating and supporting me whenever necessary during the course of my journey to acquire the Internship at AugsenseLab Pvt Ltd, TrestPark.

ABSTRACT

The development of a Django-based web application designed to monitor the real-time status of receivers by interfacing with a PostgreSQL database. The system is structured around two key database tables: one (tblStatus) that logs the status of each receiver and another (tblStatustype) that categorizes these statuses into types, such as active or inactive. The web application visually represents the operational state of each receiver using color-coded icons—green for active and red for inactive—providing users with a clear and immediate understanding of system health.

The application is built with an emphasis on real-time updates, ensuring that the displayed status of each receiver is always current. This is particularly important in environments where system uptime and quick response times are critical. The interface is designed to be user-friendly, allowing even non-technical users to efficiently monitor multiple receivers.

The result is a robust monitoring tool that enhances operational efficiency by providing immediate feedback on the status of communication receivers, ultimately contributing to improved system management and reduced downtime.

Contents

List of Figures

Chapter 1

INTRODUCTION

In modern communication systems, monitoring the status of receivers is crucial for ensuring uninterrupted service and optimal performance. This project aims to develop a Django-based web application that provides real-time monitoring of receivers' statuses using data stored in a PostgreSQL database. By displaying status information visually, the system enhances the user's ability to quickly assess and respond to any issues, improving overall system reliability.

1.1 General Background

The need for reliable monitoring systems has become increasingly critical in the era of digital communication and networked devices. As communication networks expand and become more complex, ensuring the continuous operation of receivers—the devices responsible for receiving and processing signals—is vital. These receivers play a crucial role in various industries, from telecommunications and broadcasting to security and industrial automation. Any disruption in their operation can lead to significant consequences, including data loss, service interruptions, and decreased operational efficiency.

Historically, monitoring the status of receivers and other networked devices was done through manual checks or proprietary software solutions. These methods often lacked real-time capabilities, were prone to delays, and required specialized knowledge to operate. As technology has advanced, the demand for automated, real-time monitoring systems has grown, leading to the development of web-based solutions that are accessible, user-friendly, and efficient.

Django, a high-level Python web framework, has emerged as a popular choice for developing such web-based applications due to its scalability, security features, and rapid development capabilities. It provides a powerful yet flexible environment for building applications that can interact with various databases, including PostgreSQL. PostgreSQL is a robust, open-source relational database management system known for its reliability, performance, and support for complex queries, making it an ideal backend for applications that require real-time data processing and retrieval.

In this project, Django is used to create a web application that interfaces with a PostgreSQL database to monitor the status of receivers in real time. The system is designed to store status information in two primary tables: ‘tblStatus‘, which logs the current status of each receiver, and ‘tblStatustype‘, which defines the possible status types (such as active or inactive). The web application then displays this information through a user-friendly interface that uses color-coded icons to indicate the operational state of each receiver.

This approach to monitoring combines the strengths of Django and PostgreSQL, resulting in a scalable, maintainable, and efficient solution that meets the demands of modern communication systems. By providing real-time visibility into receiver statuses, the system helps users quickly identify and address issues, thereby improving overall system reliability and reducing the risk of downtime.

1.2 Ojectives of the Project

- To design a Django-based web application for monitoring the status of receivers.

- To integrate PostgreSQL as the backend database for storing and retrieving receiver status data

- To create a visual representation of receiver statuses using color-coded icons (green for active, red for inactive).

- To implement a real-time status update mechanism that ensures the web application reflects the current state of each receiver.

- To validate the application through testing and ensure its functionality and reliability.

1.3 Movtivation of the Project

The need for real-time monitoring of communication systems is becoming increasingly critical as the complexity and scale of networks grow. Ensuring that receivers remain active and functional is vital to maintaining service quality and preventing disruptions. By developing a web-based monitoring tool, this project seeks to provide a user-friendly interface that simplifies the task of monitoring and managing receivers.

Chapter 2

LITERATURE SURVEY

2.1 Literature Survey

2.1.1 Citus: Scaling PostgreSQL

Citus extends PostgreSQL to support distributed databases by distributing data and queries across multiple servers. This allows PostgreSQL to handle workloads that exceed the capabilities of a single server, making it suitable for high-traffic applications and large-scale data operations.

Key Points:

- **Distributed Query Processing:** Citus distributes queries across multiple nodes, parallelizing the execution to improve response times and handle complex queries more efficiently.

- **Data Sharding:** The extension implements data sharding, which splits large datasets into smaller, manageable pieces distributed across the cluster. This helps in balancing the load and scaling horizontally.

- **Transactional Support:** Citus provides support for distributed transactions, ensuring consistency and reliability across multiple nodes, which is crucial for maintaining data integrity in distributed systems.

- **Automatic Rebalancing:** The system can automatically rebalance data and workloads across nodes to accommodate changes in data volume and query patterns, enhancing overall system performance and reliability.

- **Advanced Indexing:** Citus supports advanced indexing techniques such as distributed indexes to speed up query execution and improve access times for large datasets.

2.1.2 Comparative Performance of PostgreSQL and MongoDB

A comparative analysis between PostgreSQL and MongoDB focuses on their performance in handling spatio-temporal data. PostgreSQL, a relational database, shows superior performance in several key areas compared to MongoDB, a NoSQL database.

Key Points:

- **Query Performance:** PostgreSQL exhibits faster query performance due to its mature query optimizer and advanced indexing mechanisms, which can efficiently handle complex queries and large volumes of data.
- **Indexing Capabilities:** PostgreSQL's indexing options, such as B-trees, GiST, and GIN, offer flexible and powerful solutions for indexing various data types, including spatial and text data.
- **Consistency and Transactions:** PostgreSQL provides robust support for ACID (Atomicity, Consistency, Isolation, Durability) transactions, which ensures high data consistency and reliability, a critical aspect for applications with complex data interactions.
- **Scalability:** While MongoDB offers horizontal scaling through sharding, PostgreSQL's Citus extension also provides horizontal scaling, allowing PostgreSQL to compete with NoSQL databases in distributed environments.
- **Operational Complexity:** MongoDB's schema-less design can simplify data storage and scaling, but PostgreSQL's structured schema and relational model can offer better data integrity and query capabilities.

2.1.3 Spatio-Temporal Data Management

Managing spatio-temporal data involves handling information that varies across both spatial and temporal dimensions. This area of database management is critical for applications like Geographic Information Systems (GIS) and real-time analytics.

Key Points:

- **Spatial Indexing:** Techniques such as R-trees, Quad-trees, and KD-trees are used to efficiently index spatial data, facilitating fast spatial queries and data retrieval.

- **Temporal Data Management:** Managing temporal data involves techniques for handling time-series data, supporting queries over time ranges, and integrating time-based indexing.

- **Data Partitioning:** Effective partitioning strategies for spatio-temporal data can improve performance by distributing data based on spatial and temporal attributes, reducing query complexity and improving access times.

- **Integration of Spatial and Temporal Data:** Advanced systems integrate spatial and temporal data management capabilities, enabling complex queries that span both dimensions, which is essential for applications requiring detailed spatio-temporal analysis.

- **Real-Time Data Processing:** Systems are increasingly focusing on real-time processing of spatio-temporal data to support applications that require immediate insights and decision-making based on live data.

2.1.4 Indexing and Performance Optimization

Indexing is a crucial aspect of optimizing database performance. Effective indexing strategies can dramatically enhance query performance, especially in large and complex datasets.

Key Points:

- **Types of Indexes:** Common indexing techniques include B-trees, which are versatile and suitable for many queries; GiST, which supports complex data types; and GIN, optimized for full-text search and array data.

- **Index Selection:** Choosing the right type of index based on the query patterns and data characteristics is essential for optimizing performance and ensuring efficient data retrieval.

- **Index Maintenance:** Indexes need to be maintained to handle data updates and changes. Efficient index maintenance strategies are crucial to minimize performance overhead and ensure that indexes remain effective.
- **Trade-offs:** While indexes can significantly improve query performance, they also introduce overhead for data insertion, deletion, and updates. Balancing these trade-offs is key to maintaining overall system performance.
- **Advanced Indexing Techniques:** Innovations such as bitmap indexes and multidimensional indexes offer additional options for optimizing specific types of queries and data models.

Chapter 3

REQUIREMENT ANALYSIS

3.0.1 Functional Requirements

1. **Receiver Status Display:**
    - The system shall retrieve and display the current status of each receiver.
    - The status shall be indicated by a green icon for active receivers and a red icon for inactive receivers.
2. **Real-Time Updates:**
    - The system shall provide real-time updates to ensure the displayed status is current.
    - The system shall automatically refresh the receiver statuses at regular intervals or upon receiving new data.
3. **Database Management:**
    - The system shall store receiver statuses in the tblStatus table and status types in the tblStatustype table.
    - The system shall allow the retrieval of status history for audit purposes.
4. **User Interface:**
    - The system shall provide a user-friendly interface accessible via a web browser.
    - The interface shall be responsive, allowing access from devices of various screen sizes.

5. **Error Handling:**

   - The system shall notify the user if there is an issue retrieving or displaying receiver status information.
   - The system shall log errors and provide relevant details for troubleshooting.

6. **Authentication and Security:**

   - The system shall require users to authenticate before accessing the status monitoring interface.
   - The system shall enforce secure password practices and support session management.

3.0.2 Non-Functional Requirements

1. **Performance:**

   - The system shall handle up to 1,000 concurrent users without significant degradation in performance.
   - The system shall respond to user actions (such as loading the status page) within 3 seconds under normal load conditions.

2. **Scalability:**

   - The system shall be scalable to accommodate future growth, such as an increased number of receivers or higher user traffic.
   - The system architecture shall allow for easy addition of new features without major redesign.

3. **Reliability:**

   - The system shall have an uptime of 99.9%, ensuring continuous availability.
   - The system shall handle network interruptions gracefully, retrying failed operations when the connection is restored.

4. **Security:**

- The system shall encrypt all data transmissions using SSL/TLS.
- The system shall store sensitive data, such as passwords, using industry-standard hashing algorithms.
- The system shall be protected against common security threats such as SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF).

5. **Usability:**
    - The system interface shall be intuitive, requiring minimal training for new users.
    - The system shall provide clear and concise feedback to user actions, with appropriate prompts and alerts.
6. **Maintainability:**
    - The system codebase shall be modular and well-documented to facilitate future maintenance and upgrades.
    - The system shall follow best practices in software development, including version control and automated testing.
7. **Compatibility:**
    - The system shall be compatible with all major web browsers.
    - The system shall support integration with third-party monitoring tools or APIs if needed.
8. **Data Integrity:**
    - The system shall ensure that all data entered, stored, and retrieved from the database is accurate and consistent.
    - The system shall implement transaction management to maintain data integrity in case of failures.

3.0.3 System Requirements

Hardware Requirements

1. **Server:**
    - Processor: Quad-core or higher.
    - RAM: 8 GB or higher.
    - Storage: 100 GB SSD or higher.
    - Network: Gigabit Ethernet connection.
2. **Client:**
    - Any modern device with internet connectivity (PC, laptop, tablet, or smartphone).
    - Minimum 4 GB RAM.
    - Supported Web Browsers: Latest versions of Chrome, Firefox, Edge, or Safari.

Software Requirements

1. **Server-Side:**
    - **Operating System:** Ubuntu 20.04 LTS or any compatible Linux distribution.
    - Python 3.8 or higher.
    - Django 4.x.
    - PostgreSQL 13 or higher.
    - Virtual Environment (venv) for Python dependencies.
2. **Client-Side:**
    - A modern web browser with JavaScript enabled.
    - HTML5 and CSS3 support.

Chapter 4

SYSTEM DESIGN

4.1 System Design

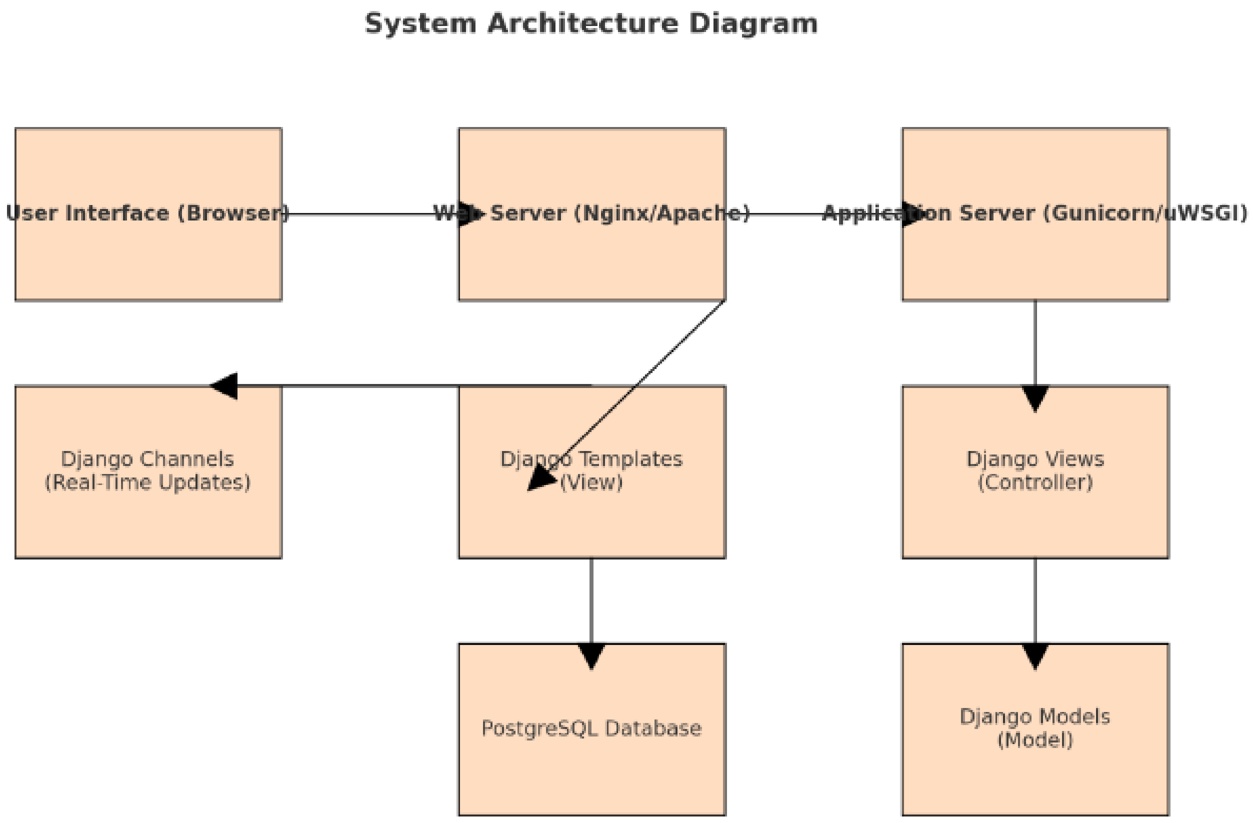
The system design for the Django-based web application monitoring receiver statuses involves several key components, including the architecture, database design, and the flow of data between the components. Below is a detailed breakdown of the system design.

4.1.1 System Architecture

The system architecture diagram for the Django-based web application. It visually represents the components involved in the system and how they interact with each other, including the user interface, web server, application server, Django components (models, views, templates), the PostgreSQL database, and the optional real-time update mechanism using Django Channels.

- **User Interface (Browser):** Where users interact with the system.
- **Web Server (Apache):** Handles HTTP requests and serves static files.
- **Application Server (Gunicorn/uWSGI)**: Runs the Django application.
- **Django Views (Controller):** Manages the business logic and processes user requests.
- **Django Models (Model):** Interacts with the PostgreSQL database to retrieve and store data.
- **Django Templates (View):** Renders HTML content to be displayed in the browser.

- **Django Channels (Real-Time Updates):** Optional component for real-time updates via WebSockets.
- **PostgreSQL Database:** Stores the receiver statuses and related data.


System Architecture Diagram
User Interface (Browser)
Web Server (Nginx/Apache)
Application Server (Gunicorn/uWSGI)
Django Channels (Real-Time Updates)
Django Templates (View)
Django Views (Controller)
PostgreSQL Database
Django Models (Model)

1. **Frontend (View):**
    - The user interface (UI) is built using HTML5, CSS3, and JavaScript.
    - It provides a responsive design to ensure the application works on various devices, including desktops, tablets, and smartphones.
    - The UI displays receiver statuses using color-coded icons (green for active, red for inactive).
    - JavaScript is used to handle real-time updates and refresh the status view dynamically.
2. **Backend (Model Controller):**

- **Django Models:** The models represent the database tables (tblStatus and tblStatustype). These models are responsible for querying and updating the PostgreSQL database.
- **Django Views:** The views handle business logic and data retrieval from the models. They process requests from the frontend and send the appropriate data back to be rendered.
- **Django Templates:** These templates render the data sent by views into HTML, which is then presented to the user.

3. **Database:**
    - **PostgreSQL:** The database stores the statuses of receivers and the status types. It is designed to be robust, scalable, and capable of handling complex queries.
4. **Web Server:**
    - **Apache:** The web server handles incoming HTTP requests, serves static files, and communicates with the Django application through Gunicorn or uWSGI.

4.1.2 Database Design

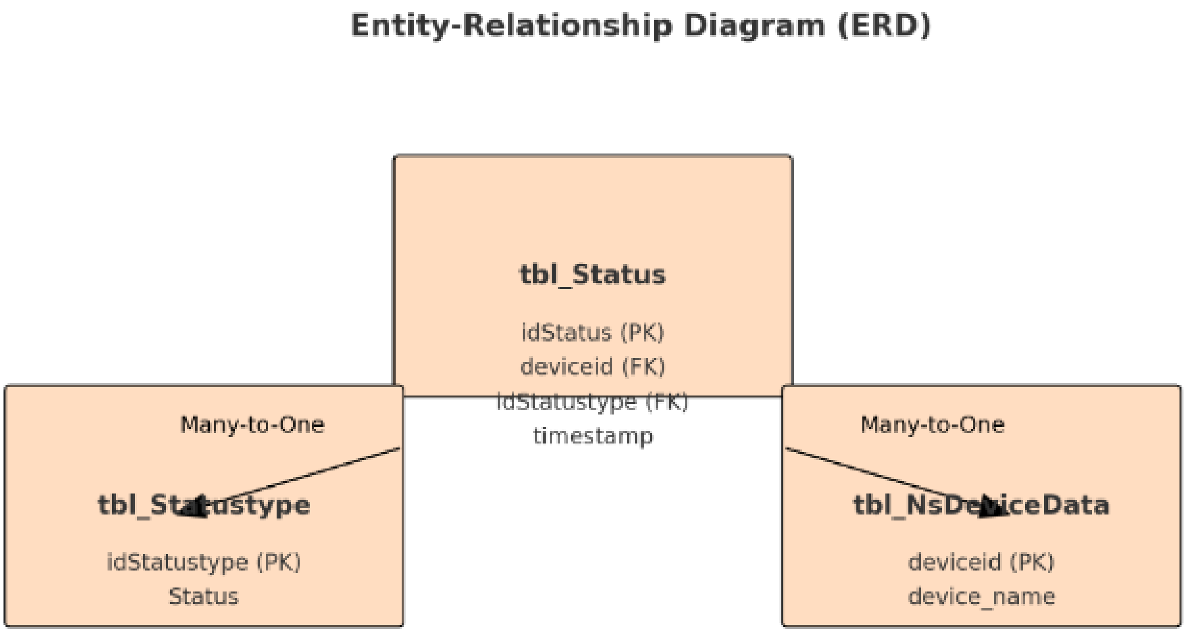
The database design involves two key tables:

1. **tblStatus:**
    - IdStatus (Primary Key, Integer): Unique identifier for each status entry.
    - Deviceid (Foreign Key, Charactervarying): References the unique identifier of the receiver from tblNsDeviceData
    - IdStatustype (Foreign Key, Integer): References the status type from tblStatustype.
    - Timestamp (timestamp with timezone): Records the exact time when the status was logged.
2. **tblStatustype:**

- IdStatustype (Primary Key, Integer): Unique identifier for each status type.
- Status (Character): Describes the type of status (e.g., 'Active', 'Inactive').

3. **tblNsDeviceData:**
    - Deviceid(Primary key,Charactervarying)
    - DeviceName(character Varying)
    - Timestamp (timestamp with time zone)
    - Latitude(float)
    - Longitude(float)
    - Altitude(float)
    - Location(character varying)

4. **ER Diagram Overview:**

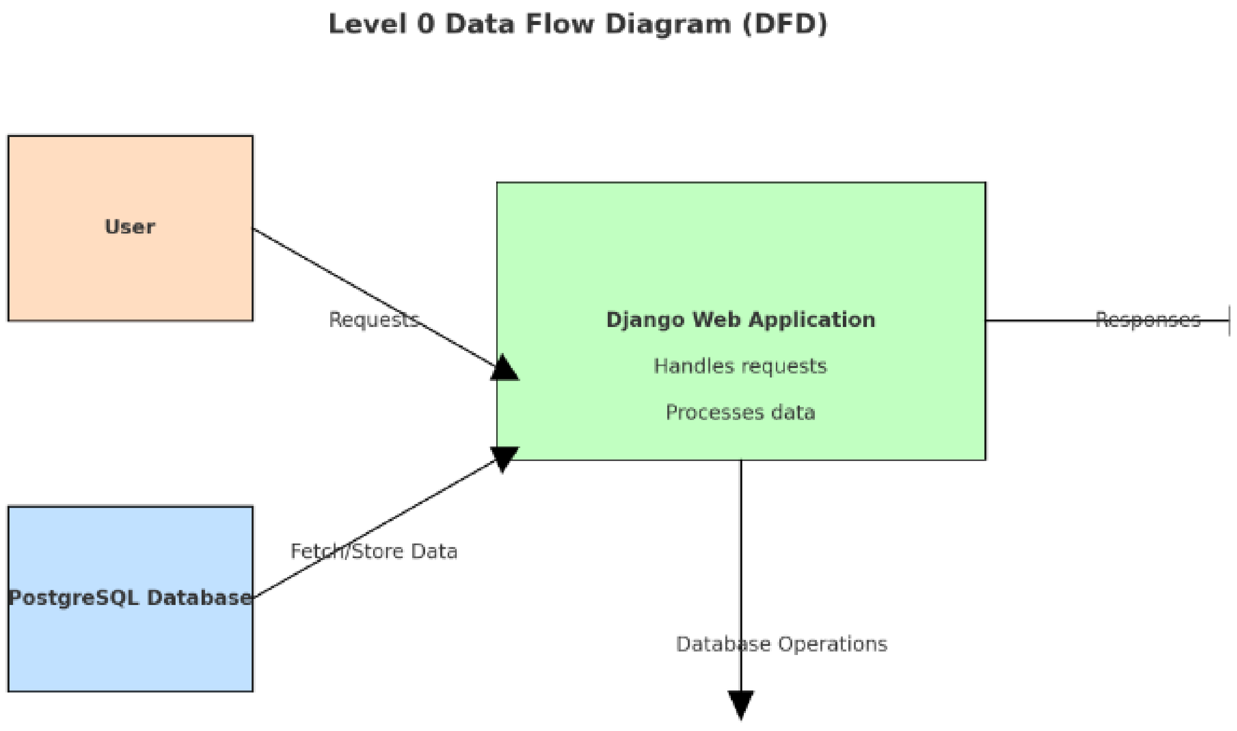


- tblStatus is related to tblStatustype through the IdStatustype foreign key.
- tblStatus is related to tblNsDeviceData through the Deviceid foreign key.

- The relationship between tblStatus and tblStatustype is many-to-one, as multiple statuses can share the same status type.
- The relationship between tblStatus and tblNsDeviceData is Many-to-one multiple status updates for a single device.

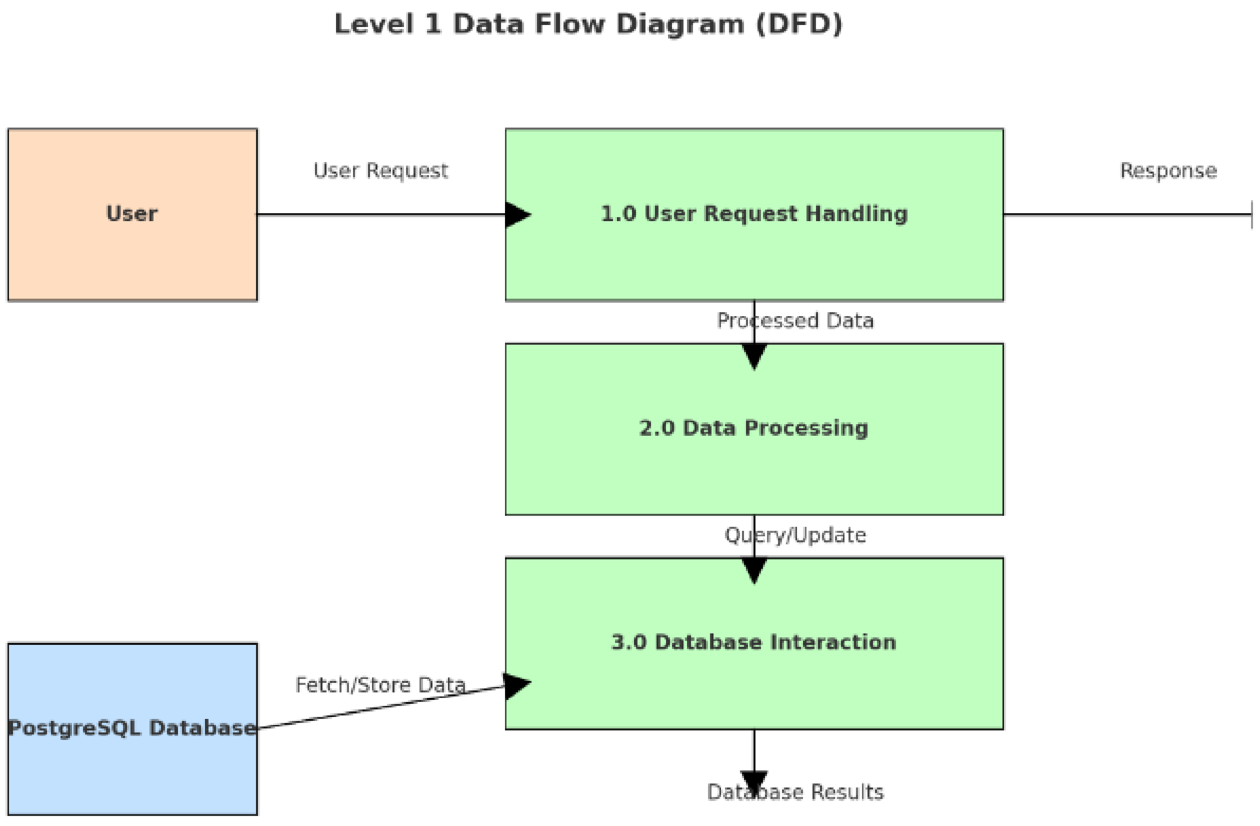
4.1.3 Data Flow Diagram

1. **Level 0: Context Diagram**
   - Users interact with the system through a web browser.
   - The web server processes requests and communicates with the Django application.
   - The Django application queries the PostgreSQL database to retrieve and update data.
   - The system sends the processed data back to the user for display.



2. **Level 1: Detailed DFD**
   - **User Input:** The user accesses the receiver status page.

- **Request Processing:** The web server forwards the request to Django views.
- **Data Retrieval:** Django views fetch the current receiver status from the PostgreSQL database using models.
- **Data Update :** Real-time data updates are pushed to the frontend using Django Channels or polling.
- **Response:** The Django templates render the data into HTML and send it back to the user's browser.
- **UI Display:** The user interface displays the statuses with corresponding color-coded icons.



4.1.4 Component Design

1. **Django Models**

    - Define the structure of the tblStatus,tblNsDeviceData tblStatustype tables.

    - Implement methods to interact with the database (e.g., fetching statuses, filtering data).

2. **Django Views**
    - StatusView: Retrieves the status of all receivers and passes the data to the template.
    - StatusDetailView: Provides detailed information about a specific receiver's status.
    - RealTimeUpdateView: Handles real-time updates and pushes new status data to the frontend.

3. **Django Templates**
    - statuslist.html: Displays the list of receivers with their statuses.
    - statusdetail.html: Shows detailed information for a specific receiver.
    - base.html: Provides a common layout, including the header, footer, and navigation.

4. **Frontend Components**
    - JavaScript: Handles real-time updates and dynamic UI interactions.
    - CSS: Styles the status icons and overall layout for a clean, user-friendly interface.
    - AJAX (Optional): Used for asynchronous data fetching without requiring full page reloads.

Chapter 5

IMPLEMENTATION

5.1 Implementation

5.1.1 Front-End Implementation

1. **User Interface Design:**

   - The user interface is designed using Django templates, which leverage HTML, CSS, and JavaScript to provide a responsive and user-friendly experience.
   - The main status monitoring page displays the status of receivers using color-coded icons (green for active, red for inactive).

2. **Responsive Design:**

   - The front-end is designed to be fully responsive, ensuring that the status monitoring dashboard is accessible and usable across various devices, including desktops, tablets, and smartphones.
   - CSS frameworks like Bootstrap or Tailwind CSS are used to streamline responsive design implementation.

3. **Template System:**

   - Django's template system is utilized to dynamically generate HTML pages based on the data retrieved from the database.
   - Templates are structured to separate presentation logic from business logic, ensuring clean and maintainable code.

4. **Real-Time Status Updates:**

- To ensure the receiver statuses are up-to-date, real-time updates are implemented using WebSockets, facilitated by Django Channels.
- The front-end automatically refreshes the status icons without requiring a page reload, providing a seamless user experience.

5.1.2 Back-End Implementation

1. **Database Design:**
   - The system's database is designed using PostgreSQL, with two main tables: `tblStatus` and `tblStatustype`.
   - `tblStatus` contains the status entries for each receiver, while `tblStatustype` categorizes the types of statuses (e.g., active, inactive).
   - `tblStatus` is related to `tblStatustype` through a foreign key, creating a many-to-one relationship.
2. **Django Models:**
   - Django models are created to map the database tables to Python objects, allowing easy manipulation of database records.
   - The models define the structure of the data and include methods for common database operations.
3. **Views and Controllers:**
   - Django views handle the business logic, processing requests from the user and returning the appropriate responses.
   - Controllers manage the interaction between the models and the views, ensuring that the correct data is passed to the templates for rendering.
   - Views are responsible for querying the database for receiver statuses and passing this data to the templates for display.
4. **Real-Time Data Handling:**
   - Real-time data handling is implemented using Django Channels, which integrates WebSocket support into the Django framework.

- The back-end listens for changes in receiver statuses and pushes updates to the front-end in real-time.
- Redis is used as the channel layer backend to manage message routing between clients and the server.

5. **Security and Authentication:**
    - The back-end implements Django's built-in authentication system to ensure that only authorized users can access and monitor the receivers.
    - User data and session management are handled securely, with options for role-based access control depending on the project requirements.
6. **Error Handling and Logging:**
    - Comprehensive error handling is implemented to manage unexpected situations, such as database connection failures or invalid user input.
    - Logging mechanisms are set up using Django's logging framework to track application performance and errors.
7. **Testing and Deployment:**
    - The application is thoroughly tested using Django's testing framework, covering unit tests, integration tests, and end-to-end tests.
    - For deployment, a combination of Nginx and Gunicorn is used to serve the Django application, ensuring scalability and performance.
    - The application is containerized using Docker to streamline deployment and ensure consistency across environments.

5.1.3 Database Design

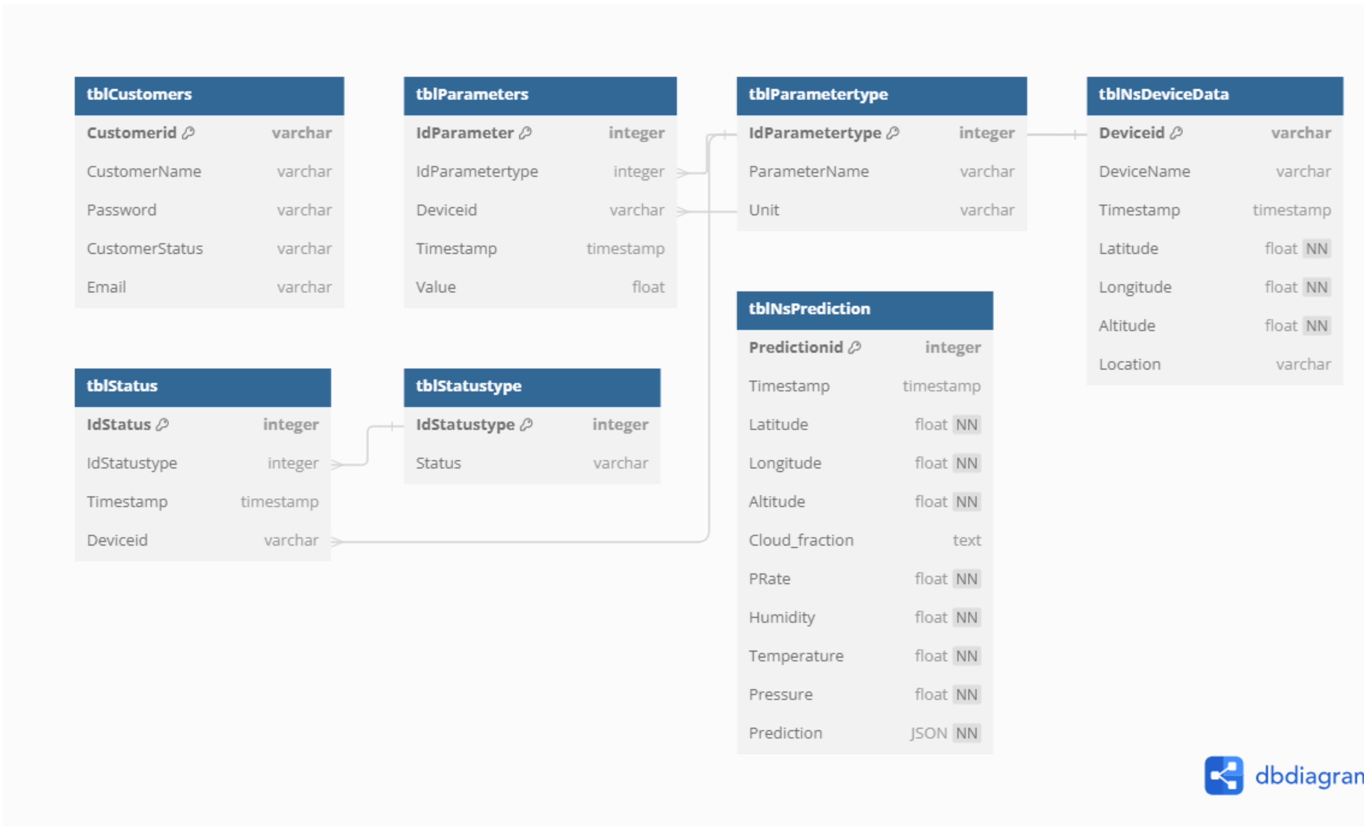
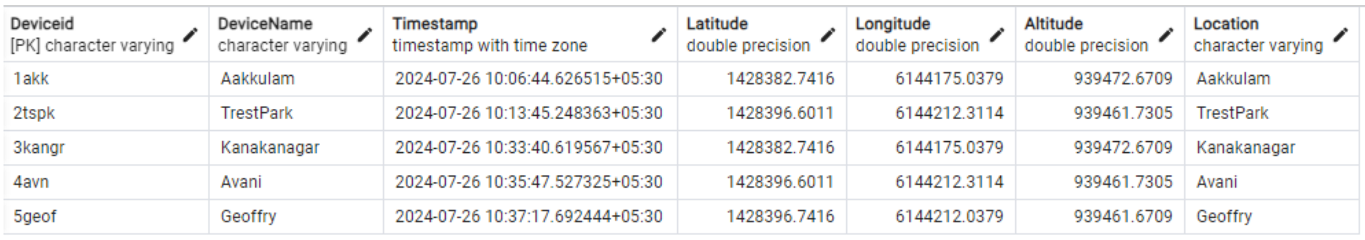


Figure 5.1: Database Design

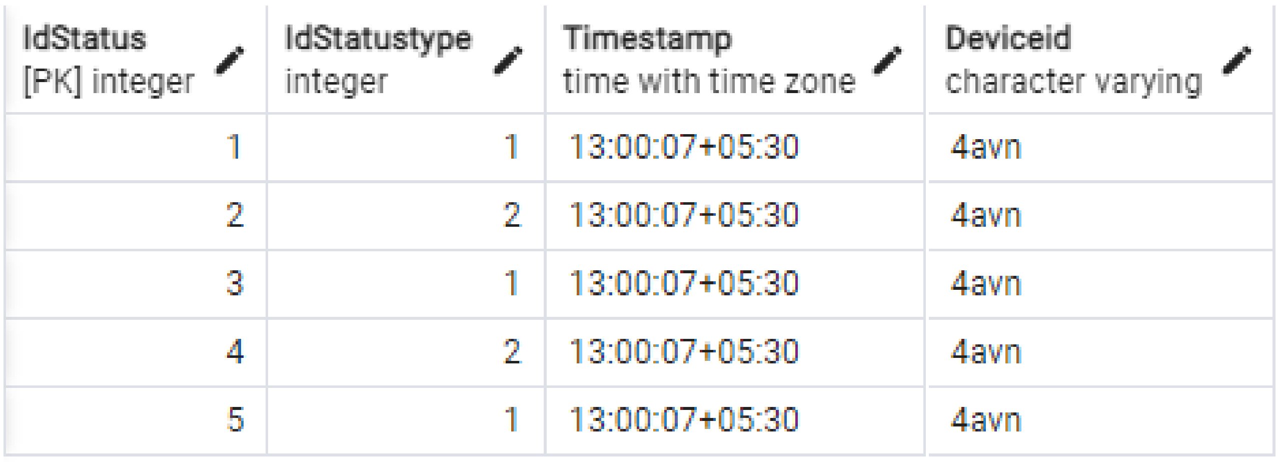
5.1.4 Database Creation and Updation

Creating the table tblNsDeviceData, tblStatus and tblStatustype and insert data into the table.



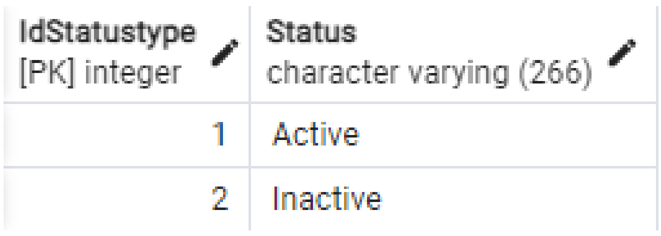
Deviceid [PK] character varying	DeviceName character varying	Timestamp timestamp with time zone	Latitude double precision	Longitude double precision	Altitude double precision	Location character varying
1akk	Aakkulam	2024-07-26 10:06:44.626515+05:30	1428382.7416	6144175.0379	939472.6709	Aakkulam
2tspk	TrestPark	2024-07-26 10:13:45.248363+05:30	1428396.6011	6144212.3114	939461.7305	TrestPark
3kangr	Kanakanagar	2024-07-26 10:33:40.619567+05:30	1428382.7416	6144175.0379	939472.6709	Kanakanagar
4avn	Avani	2024-07-26 10:35:47.527325+05:30	1428396.6011	6144212.3114	939461.7305	Avani
5geof	Geoffry	2024-07-26 10:37:17.692444+05:30	1428396.7416	6144212.0379	939461.6709	Geoffry

Figure 5.2: tblNsDeviceData



IdStatus [PK] integer	IdStatustype integer	Timestamp time with time zone	Deviceid character varying
1	1	13:00:07+05:30	4avn
2	2	13:00:07+05:30	4avn
3	1	13:00:07+05:30	4avn
4	2	13:00:07+05:30	4avn
5	1	13:00:07+05:30	4avn

Figure 5.3: tblStatus



IdStatustype [PK] integer	Status character varying (266)
1	Active
2	Inactive

Figure 5.4: tblStatustype

Chapter 6

RESULTS

6.1 Results

6.1.1 Real-Time Status Monitoring

- The application successfully displays the real-time status of receivers based on data retrieved from the PostgreSQL database. The status is visually represented using color-coded icons: green for active receivers and red for inactive receivers.

- The real-time updates ensure that the status of each receiver is always current, with the application able to reflect changes in status almost instantaneously.

- User interactions with the interface are smooth and responsive, allowing users to easily monitor the operational status of multiple receivers simultaneously.

6.1.2 Database Interaction Performance

- The integration between the Django web application and the PostgreSQL database proved to be robust and efficient. Database queries for retrieving and updating receiver statuses were executed quickly, even with a large dataset.

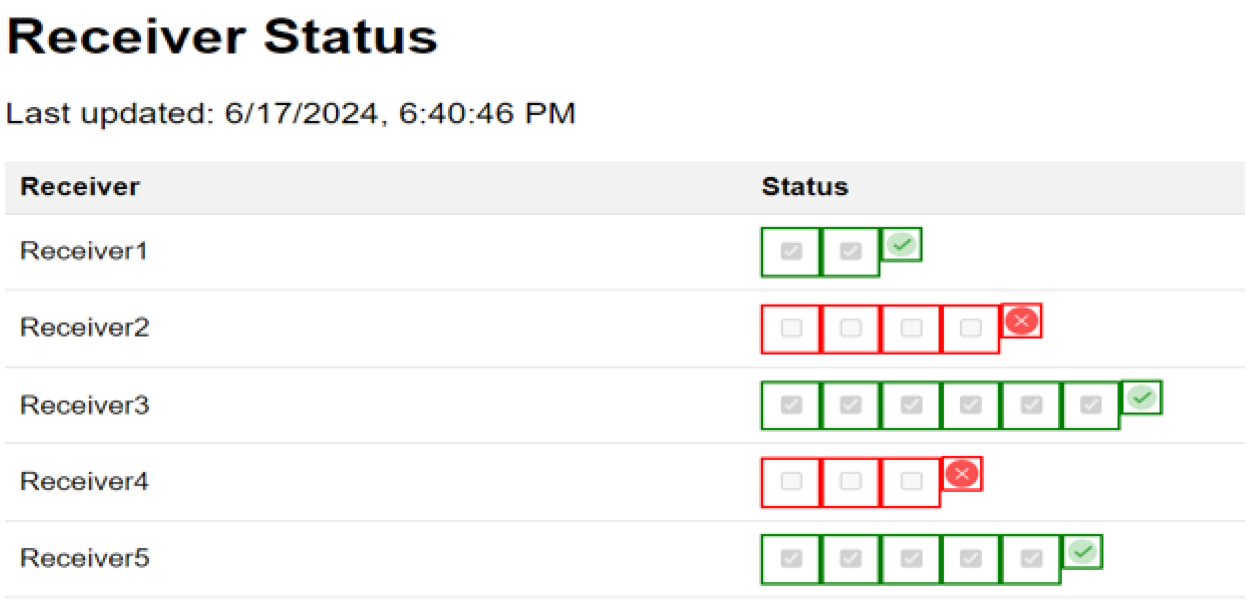
- The system's performance was consistent under various loads, demonstrating scalability and reliability. No significant performance degradation was observed during peak usage.

- The relational integrity between the `tbl_Status` and `tbl_Statustype` tables was maintained, ensuring accurate status categorization and display.

6.1.3 User Experience and Interface Usability

- User feedback highlighted the simplicity and clarity of the user interface, with the color-coded status indicators being particularly effective in conveying the status of receivers at a glance.
- The application was tested across various devices and screen sizes, ensuring that the interface was responsive and accessible on desktops, tablets, and smartphones.
- The application's design and functionality received positive reviews for being intuitive and requiring minimal training for new users.

6.1.4 Status Page



Receiver Status

Last updated: 6/17/2024, 6:40:46 PM

Receiver	Status
Receiver1	☑ ☑ ✓
Receiver2	☐ ☐ ☐ ☐ ✗
Receiver3	☑ ☑ ☑ ☑ ☑ ☑ ✓
Receiver4	☐ ☐ ☐ ✗
Receiver5	☑ ☑ ☑ ☑ ☑ ✓

Figure 6.1: Status page

Chapter 7

CONCLUSION

7.1 Conclusion

To effectively display receiver statuses on a webpage using Django, the implementation involves a few key steps to ensure clarity and functionality. First, you need to design a webpage layout that includes a header and a list of receivers, each accompanied by a status indicator. The status indicator is visually represented by a small, colored circle: green for active and red for inactive receivers. This visual distinction helps users quickly assess the operational status of each receiver.

The implementation involves setting up Django models to represent the statuses and their types, then creating a Django view to fetch and pass this data to a template. In the template, you use CSS to style the status indicators appropriately. If preferred, these indicators can be designed as images using graphic tools, but simple CSS styling can achieve similar results with less complexity.

In practice, you would render a webpage where each receiver's status is shown with a clear, colored icon next to its identifier. This design approach ensures that users can easily interpret the status of each receiver at a glance, enhancing the user experience and providing an intuitive interface for monitoring device statuses.

Chapter 8

FUTURE WORK

8.1 Future Work

- **Enhanced Status Indicators:** Future work could involve expanding the range of status indicators beyond just active and inactive. For example, additional statuses could include "maintenance," "error," or "warning," each with its own color code, to provide more granular monitoring.

- **Notification System:** Implementing a notification system that alerts users via email or SMS when a receiver changes status could further enhance the system's utility, especially in critical environments where immediate response is required.

- **Analytics and Reporting:** Adding analytics and reporting features could provide users with insights into receiver performance over time. This could include generating reports on the frequency of status changes, average downtime, or overall system health.

- **Mobile Application Integration:** Developing a companion mobile application could make it easier for users to monitor receiver statuses on the go, ensuring that critical information is always accessible.

- **Machine Learning for Predictive Maintenance:** Incorporating machine learning algorithms could allow the system to predict potential receiver failures based on historical data, enabling proactive maintenance and reducing downtime.

Chapter 9

REFERENCES

9.1 References

[1] Cubukcu, Umur, et al. ”Citus: Distributed postgresql for data-intensive applications.” Proceedings of the 2021 International Conference on Management of Data. 2021.

[2] Makris, Antonios, et al. ”Performance Evaluation of MongoDB and PostgreSQL for Spatio-temporal Data.” EDBT/ICDT Workshops. 2019.