**Jose Rizal Memorial State University**

**Gov. Guading Adaza St., Sta. Cruz,**

**Dapitan City, Philippines**

**GSU Gateway Technical Documentation**

By:

Edrean Balonga

Bernadette Bantillan

Bryan Bantillan

Patrick Joshua Nicholas

Jenifer Saguindang

May 19, 2025

**General Service Office**

**Table of Contents**

**I. Overview 1**

**II. Workflow**

**User Case Diagram 2-3**

**GSU Gateway Flowchart 4-5**

**Service Request Workflow 6**

**Notifications Workflow 6**

**Account Creation Workflow 6-7**

**III. Architecture Decision Record (ADR)**

**Technologies 8-9**

**Rationale for Choosing these Technologies 9-10**

**Alternatives Considered and Rejected 10-11**

**Consequences 11-12**

**IV. Database Design 13-17**

**V. Error Handling 18-19**

**VI. Security 20-21**

**VII. Conclusion 22-23**

**I. Overview**

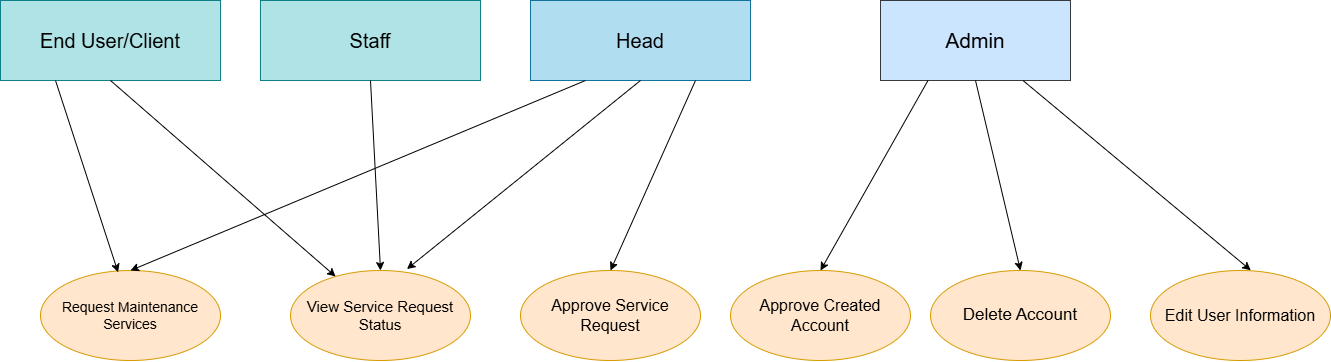
GSU Gateway is an online ticketing platform designed to streamline and automate service requests within the campus General Service Unit (GSU). Originally, the GSU services were categorized into three parts: Maintenance Service Request, Transportation Service Request, and Venue Reservation Request (for event spaces and facilities). However, for the initial phase of development, the team focused solely on implementing the Maintenance Service Request module.

This system enables clients—such as faculty and staff—to submit requests for maintenance services, including janitorial, carpentry, electrical, and air conditioning services. By transitioning from traditional paper-based processes to an online system, the GSU Gateway aims to reduce paper consumption, aligning with sustainable practices and contributing to environmental conservation.

This digital approach not only minimizes paper waste but also enhances operational efficiency by automating the service request process, thereby reducing manual handling and expediting response times. Clients can submit service requests through an intuitive online interface, detailing the type of service required and any pertinent information. Each request undergoes a verification process by the staff, followed by an approval process handled by the Campus Director and the Head of GSU. Once approved, the system notifies both the client and assigned personnel, who then coordinate the task, finalize the schedule, and assign a priority number. After the service is completed, clients provide feedback, ensuring continuous improvement in service quality.

By automating request submission, approval, and scheduling, GSU Gateway enhances efficiency, transparency, and communication, replacing manual processes with a faster and more organized digital system.

**II. Workflow**



**Figure 1. User Case**

The **Use Case Diagram** represents the key functionalities of the **GSU Gateway System** and the interactions between different user roles: **End User/Client, Staff, Head, and Admin**.

### **User Roles and Use Cases:**

#### ****1. *End User / Client*****

* **Use Case: Request Maintenance Services**
  + **Description:** Allows end users or clients to submit maintenance service requests to the system. This action initiates the maintenance workflow.

#### ****2. *Staff*****

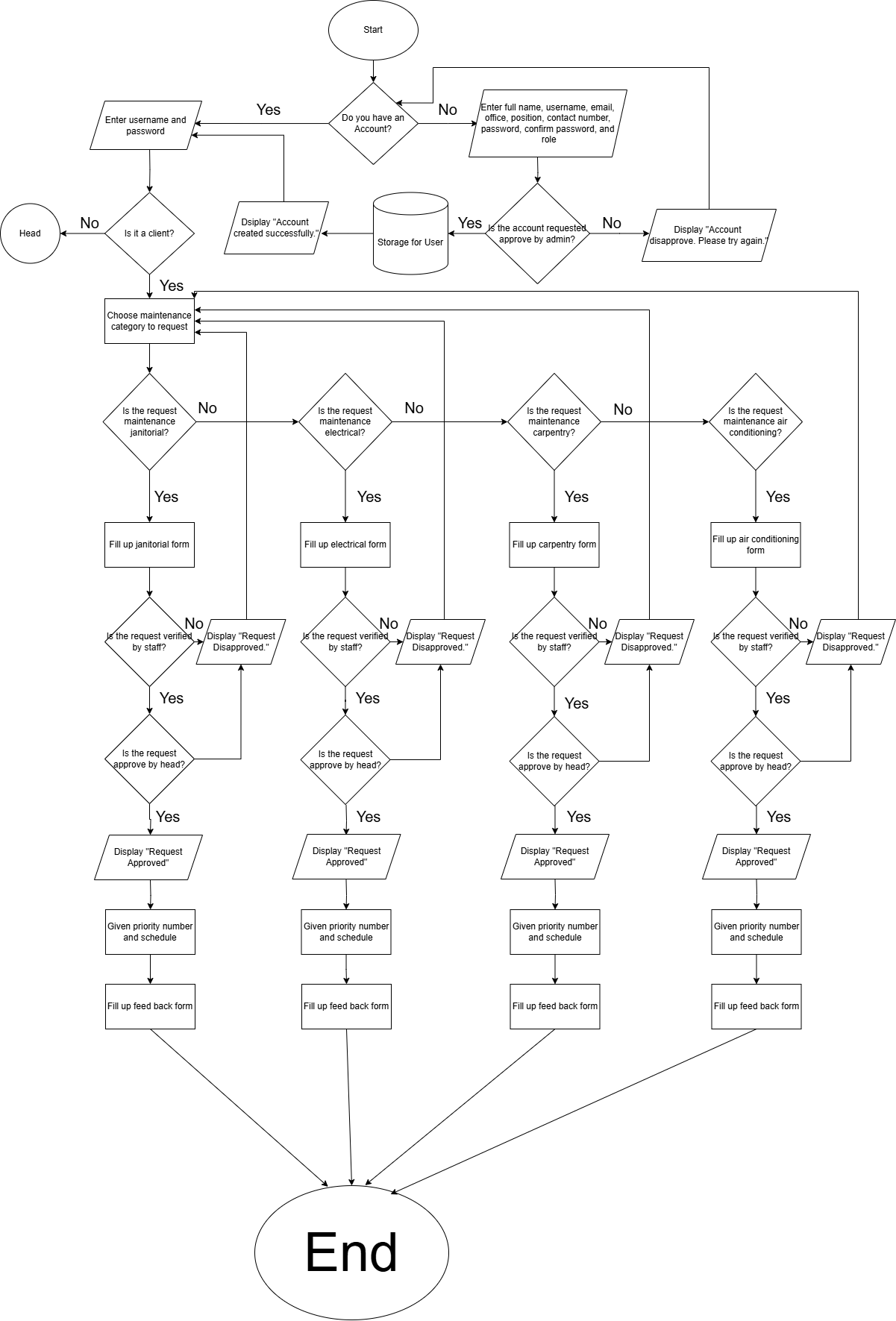
* **Use Case: View Service Request Status**
  + **Description:** Staff members can track the status of service requests submitted by end users. This provides visibility on request progress and updates.

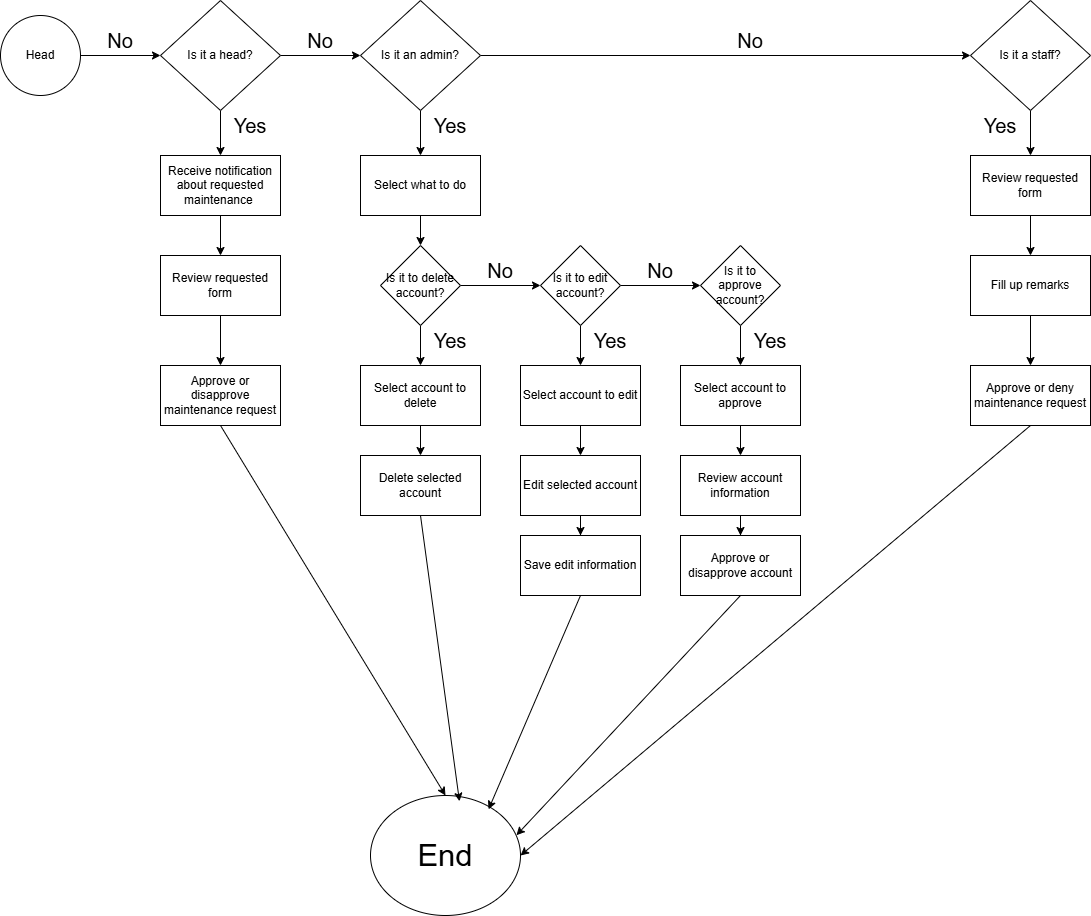
#### ****3. *Head*****

* **Use Case: Approve Service Request**
  + **Description:** The department head is responsible for reviewing and approving or rejecting submitted service requests. This step is critical for ensuring appropriate prioritization and authorization.

#### ****4*. Admin*****

* **Use Case: Approve Created Account**
  + **Description:** Admins validate and approve new user accounts before granting access to the system.
* **Use Case: Delete Account**
  + **Description:** Admins can remove existing user accounts from the system when they are no longer needed or violate policies.
* **Use Case: Edit User Information**
  + **Description:** Admins can update user information such as name, role, or department affiliation, maintaining accurate and up-to-date user data.

**Figure 2. GSU Gateway Flowchart Part I**

****

**Figure 3. GSU Gateway Flowchart Part 2**

**4.1 Service Request Workflow**

1. **User Submission**: The process begins when a client submits a request via the frontend. The request includes service details, which is validated by the backend before being stored in the database.

2. **Approval Process**: The Campus Director and GSU are notified of the request via an internal dashboard. They review the request and either approve or deny it. Once a decision is made, the system updates the request status (approved, denied, pending) in the database.

3. **Service Fulfillment**: After approval, the relevant service personnel are assigned to fulfill the request. The system updates the request status to indicate when the service is in progress or completed.

4. **Feedback Collection**: After the service is completed, the client receives a notification requesting them to provide feedback. The client submits their feedback through the frontend, which is stored in the database for future analysis.

**4.2 Notification Workflow**

Notifications are a key part of the system to ensure all stakeholders are informed. When a client submits a service request, they receive a confirmation notification via email, which is the method currently integrated into the system. The Campus Director and GSU are also alerted about new requests through email. Once a request is approved or rejected, the client is notified by email as well. Similarly, after service completion, clients are reminded via email to submit their feedback. Although the office suggested sending notifications through SMS or messaging due to internet connectivity issues, the team found this challenging to implement. Therefore, all notifications continue to be sent in real-time through the system’s email notification services.

### **4.3 Account Creation Workflow**

The GSU Gateway system originally integrated a user-driven registration workflow. However, based on panelist recommendations, the final implementation follows an **admin-controlled account creation process**.

#### 4.3.1 Original Design: User Self-Registration

1. **User Registration:**  
   New users could access the frontend and create an account by entering:
   * Full Name
   * Username
   * Email
   * Office
   * Position
   * Contact Number
   * Password and Confirm Password
   * Role selection (Client, Staff, Head, Admin)
2. **Account Storage:**  
   Upon submission, the backend validated the information. If valid, the account was temporarily stored pending admin approval.
3. **Admin Approval:**
   * If approved, users were notified and granted access to log in.
   * If rejected, users were prompted to correct and resubmit their registration details.

#### 4.3.2 Final Design (Post-Panelist Recommendation): Admin-Only Account Creation

* **Account Creation:**  
  Only the system administrator can create user accounts through the admin interface.
* **One Account Per Department:**  
  Each department is assigned only **one account**, either for the Dean or an officially designated department representative.
* **Assigned Roles:**  
  The admin assigns the proper role upon account creation (Client, Staff, Head, Admin).
* **Benefits:**
  + Improved security.
  + Streamlined approvals.
  + Accountability and traceability per department.

**III. Architecture Decision Record (ADR)**

**Context**

To build an efficient, scalable, and maintainable front-end, ensure comprehensive testing, facilitate documentation and deployment, streamline backend development, and optimize project management, our team has chosen specific technologies based on performance, ease of use, community support, and industry standards.

**Technologies**

The team will use the following technologies for front-end development, quality assurance (QA) testing, backend development, documentation and deployment, and project management:

**Front-End Development**

* **React.js**: A component-based JavaScript framework that improves performance using a virtual DOM.
* **Tailwind CSS**: A utility-first CSS framework that speeds up development and ensures design consistency.
* **Visual Studio Code (VS Code)**: A powerful and lightweight code editor.
* **Figma**: A collaborative wireframing and UI design tool.

**QA Testing**

* **Selenium**: For UI automation testing to ensure consistent user interface behavior.
* **Postman**: For API testing to verify backend communication.
* **OWASP ZAP**: For security testing to detect vulnerabilities early.
* **JMeter**: For performance testing to simulate high loads and analyze system performance.

**Backend Development**

* **PHP**: A widely used server-side scripting language known for its simplicity, flexibility, and extensive community support. It integrates well with MySQL, making it ideal for dynamic web applications requiring database interactions. PHP is also easy to deploy on various hosting providers and works seamlessly across different operating systems.
* **XAMPP**: A lightweight, open-source software package that provides a local development environment, including Apache, MySQL, PHP, and Perl. It simplifies server configuration, allowing developers to test their applications locally before deployment.
* **VS Code**: A powerful and lightweight code editor that supports PHP development with extensions like PHP Intelephense and XDebug. It offers essential features such as syntax highlighting, debugging tools, and Git integration, making PHP development more efficient.

**Documentation and Deployment**

* **GitHub:** A collaborative platform that allows users to store, share, and work together on code.
* **Microsoft Word:** A widely used documentation tool that allows structured documentation, formatting, and collaboration for technical and non-technical documentation needs.
* **Netlify:** Chosen for front-end deployment due to its simple setup, continuous deployment capabilities, automatic build process, free SSL, serverless functions support, and global CDN for fast website delivery. Netlify simplifies frontend hosting with a user-friendly interface and quick GitHub integration, significantly speeding up the deployment workflow.
* **Railway:** Used for backend and database deployment because of its seamless provisioning of backend services and databases, straightforward environment configuration, automatic scaling, and generous free tier. Railway's intuitive UI and strong GitHub integration make backend deployments faster and easier compared to traditional cloud providers.
* **Docker:** Implemented for containerized deployment of backend applications to ensure consistency across development, testing, and production environments. Docker simplifies deployment by packaging the application and all its dependencies into a portable container. This eliminates the "works on my machine" problem and enhances scalability, flexibility, and portability across different cloud providers or local environments.

**Project Management**

* **Trello**: A popular project management tool that offers a unique approach to organizing and managing tasks and projects.

**Rationale for Choosing These Technologies**

**Front-End:**

* **React.js** is chosen due to its large community, reusable component-based structure, and optimized performance via virtual DOM.
* **Tailwind CSS** allows rapid UI development, prevents unnecessary CSS overrides, and ensures a consistent design.
* **VS Code** is widely used, supports extensive extensions, and enhances developer productivity.
* **Figma** enables efficient wireframing, collaboration, and design consistency.

**QA Testing:**

* **Selenium & JMeter** provide automation, reducing manual effort and increasing test efficiency.
* **OWASP ZAP** enhances security by detecting vulnerabilities at early stages.
* **Postman** simplifies API testing and ensures smooth backend interactions.
* These tools are industry standards, well-documented, and supported by a large community.

**Backend Development:**

* **PHP** is widely adopted, easy to deploy, and integrates well with MySQL.
* **XAMPP** simplifies local development and server configuration.
* **VS Code** provides powerful extensions for PHP development, enhancing productivity.

**Documentation and Deployment:**

* **GitHub** is used due to its widespread adoption, robust community, and strong integration capabilities.

**Project Management:**

* **Trello** offers a visual interface and flexibility that set it apart, making task management intuitive and efficient.

**Alternatives Considered and Rejected**

**Front-End Alternatives:**

* **Vue.js**: While an alternative framework, it has a steeper learning curve for certain use cases and a smaller community than React.js.
* **SCSS**: Offers styling flexibility but increases complexity and may lead to redundant styles compared to Tailwind CSS.

**QA Testing Alternatives:**

* **Cypress**: While great for UI testing, it lacks strong cross-browser support compared to Selenium.
* **Katalon Studio**: A powerful tool but requires licensing for advanced features, making it less cost-effective.

**Backend Development Alternatives:**

* **Django**: Though powerful, it requires learning Python, making the transition challenging.
* **Node.js**: Efficient but requires JavaScript for backend development, which may not be ideal for teams experienced in PHP.
* **Ruby on Rails**: Has a steeper learning curve and fewer hosting options.

**Documentation and Deployment Alternatives:**

* **GitLab:** More resource-intensive for self-hosting and has a steeper learning curve for some teams.
* **Bitbucket:** Primarily suited for teams using Jira and may not be as popular as GitHub.
* **Azure DevOps:** Heavily integrated with Microsoft products, which may not be ideal for teams outside the Azure ecosystem.
* **AWS CodeCommit:** Best for AWS-based workflows but lacks the strong developer community and marketplace that GitHub provides.
* **Google Cloud Source Repositories:** Limited adoption and features compared to GitHub.
* **Vercel (Frontend Alternative):** Optimized especially for Next.js apps and offers similar benefits to Netlify; however, for general frontend deployment without a Next.js dependency, Netlify provides broader use case support and slightly easier setup for static sites and non-Next.js projects.
* **GitHub Pages (Frontend Alternative):** Great for static sites but lacks advanced features like serverless functions and automatic form handling that Netlify offers.
* **Render (Backend/Database Alternative):** A strong competitor to Railway offering easy backend deployment, but Railway offers a cleaner interface and more streamlined provisioning for small to medium-sized applications.
* **Heroku (Backend/Database Alternative):** Was a popular choice for backend and database hosting; however, its free tier was recently limited, making it less attractive compared to Railway, which continues to offer more for free.
* **Podman (Containerization Alternative):** Podman is a rootless, daemonless container engine that provides an alternative to Docker for container management. However, Docker remains more widely adopted, has stronger community support, and offers a richer ecosystem of tools and integrations, making it a better fit for our deployment needs.

**Project Management Alternatives:**

* **Jira**: Too complex for non-software teams.
* **Asana**: Overwhelming with too many features.
* **Monday.com**: Requires more customization and setup.
* **ClickUp**: Steeper learning curve and potential complexity.
* **Microsoft Planner**: Limited flexibility and integrations.

**Consequences**

**Positive Impact:**

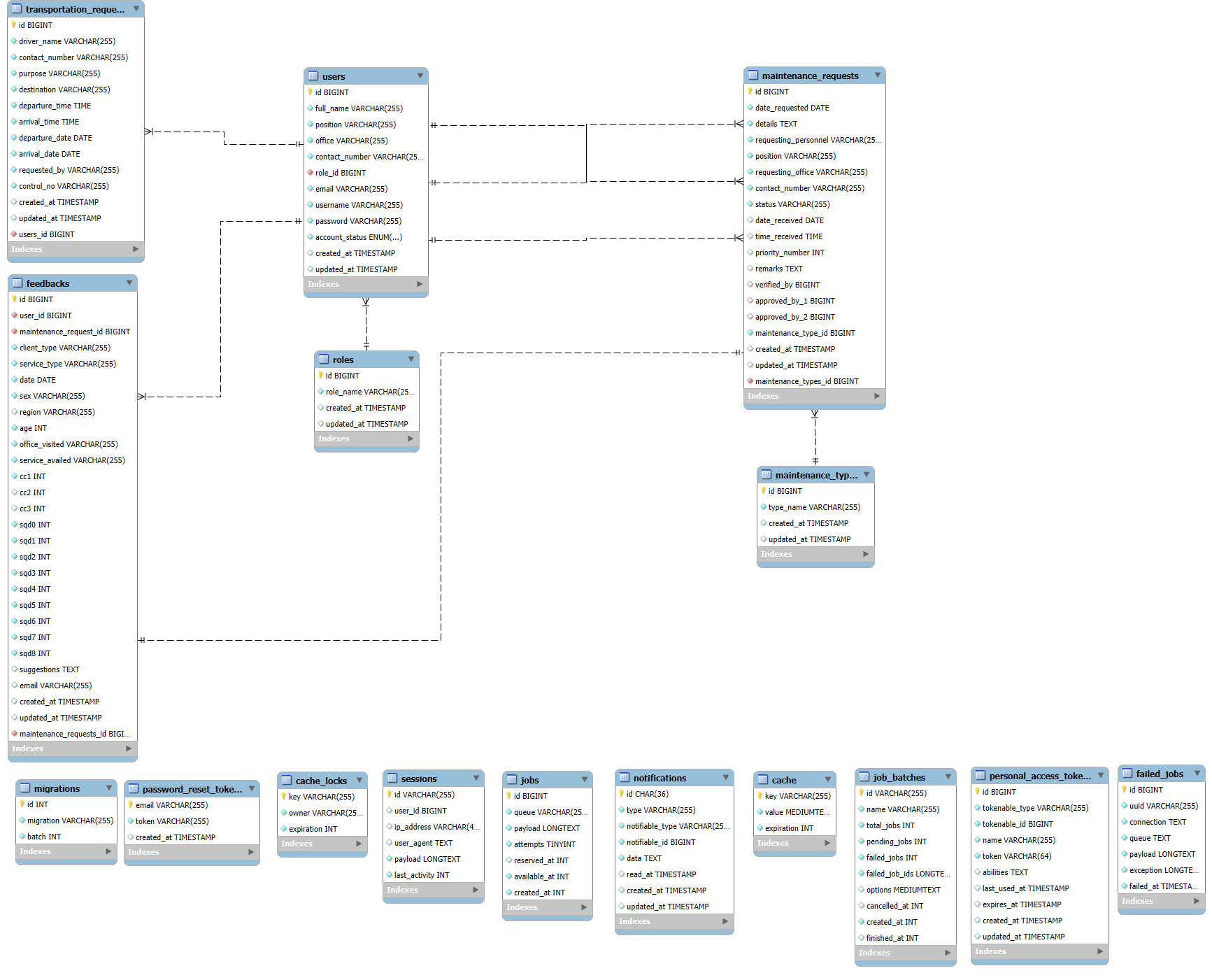
* The chosen stack will ensure faster development, better maintainability, and high-performance UI.
* Testing tools will cover UI, API, security, and performance, leading to a robust and reliable system.
* GitHub will enhance collaboration and streamline documentation and deployment processes.
* Trello will improve project organization and task tracking.
* PHP and XAMPP will provide a flexible and efficient backend development environment.
* GitHub strengthens team collaboration through version control and seamless integration with deployment platforms.
* Netlify makes front-end deployment extremely easy with automatic GitHub integration, free SSL, and global CDN support.
* Railway simplifies backend and database deployment with minimal setup and smooth GitHub integration.
* Docker ensures consistent behavior across different environments, enhances scalability, and improves portability.

**Potential Trade-offs:**

* React.js has a learning curve, but the large community and documentation ease adoption.
* Tailwind CSS requires developers to adapt to a utility-based approach, but it significantly reduces styling conflicts.
* QA automation tools require setup and maintenance, but they save time in the long run.
* GitHub may require teams to adapt to version control best practices.
* Trello may not be as feature-rich for agile development compared to Jira.
* PHP may not be as modern as newer backend technologies, but it remains widely supported.
* Netlify is less suited for server-heavy applications and the free tier can become costly for larger projects.
* Railway apps on free plans may "sleep," causing cold starts, and scaling for larger systems may require migration.
* Docker introduces setup complexity and demands higher CPU and memory resources during development.

**IV. Database Design**

The GSU Gateway database is designed to store, organize, and manage information related to service requests within the campus General Service Unit (GSU). It supports critical operations such as request creation, verification, approval, personnel assignment, scheduling, and feedback collection. The system database ensures efficiency, transparency, and traceability of service workflows by maintaining structured records of all transactions.

**

**Figure 4: Entity-Relationship Diagram for GSU Gateway**

The ERD above depicts the core entities, their attributes, and relationships that support the GSU Gateway’s operations.

***Entity Descriptions***

### *1. Users*

| **Field** | **Data Type** | **Description** |
| --- | --- | --- |
| id | BIGINT | Primary key. |
| name | VARCHAR(255) | Full name of the user. |
| position | VARCHAR(255) | Position of the user (e.g., faculty, staff, personnel). |
| office | VARCHAR(255) | Office or department affiliation. |
| contact\_number | VARCHAR(255) | User's phone number. |
| email | VARCHAR(255) | User's email address. |
| password | VARCHAR(255) | Encrypted user password. |
| role\_id | BIGINT | Foreign key linking to **roles** table. |
| created\_at, updated\_at | TIMESTAMP | Timestamps for record creation and update. |

### *2. Roles*

| **Field** | **Data Type** | **Description** |
| --- | --- | --- |
| id | BIGINT | Primary key. |
| role\_name | VARCHAR(255) | Role of the user (e.g., Client, Staff, Director, Head, Personnel). |
| created\_at, updated\_at | TIMESTAMP | Timestamps for record creation and update. |

### *3. Maintenance Requests*

| **Field** | **Data Type** | **Description** |
| --- | --- | --- |
| id | BIGINT | Primary key. |
| date\_requested | DATE | Date when the request was made. |
| details | TEXT | Description of the maintenance request. |
| requesting\_personnel | VARCHAR(255) | Name of requester. |
| requesting\_office | VARCHAR(255) | Office of the requester. |
| contact\_number | VARCHAR(255) | Contact info of requester. |
| status | VARCHAR(255) | Status of request (Pending, Verified, Approved, Completed, etc.). |
| date\_received | DATE | Date when request was received. |
| time\_received | TIME | Time when request was received. |
| priority\_number | INT | Priority level assigned after approval. |
| remarks | TEXT | Additional remarks. |
| verified\_by, approved\_by\_1, approved\_by\_2 | BIGINT | Foreign keys to **users** table (staff, director, head). |
| maintenance\_type\_id | BIGINT | Foreign key to **maintenance\_types** table. |
| created\_at, updated\_at | TIMESTAMP | Timestamps for record creation and update. |

### *4. Maintenance Types*

| **Field** | **Data Type** | **Description** |
| --- | --- | --- |
| id | BIGINT | Primary key. |
| type\_name | VARCHAR(255) | Type of maintenance (e.g., Janitorial, Carpentry, Electrical, Air Conditioning). |
| created\_at, updated\_at | TIMESTAMP | Timestamps for record creation and update. |

### *5. Transportation Requests*

| **Field** | **Data Type** | **Description** |
| --- | --- | --- |
| id | BIGINT | Primary key. |
| driver\_name | VARCHAR(255) | Name of the driver assigned. |
| contact\_number | VARCHAR(255) | Contact number. |
| purpose | VARCHAR(255) | Purpose of the trip. |
| destination | VARCHAR(255) | Destination details. |
| departure\_time | TIME | Time of departure. |
| departure\_date | DATE | Date of departure. |
| arrival\_time | TIME | Time of arrival. |
| arrival\_date | DATE | Date of arrival. |
| requested\_by | VARCHAR(255) | Requesting user’s name. |
| created\_at, updated\_at | TIMESTAMP | Timestamps for record creation and update. |

### *6. Feedbacks*

| **Field** | **Data Type** | **Description** |
| --- | --- | --- |
| id | BIGINT | Primary key. |
| user\_id | BIGINT | Foreign key linking to the **users** table. |
| maintenance\_request\_id | BIGINT | Foreign key linking to the **maintenance\_requests** table. |
| service\_type | VARCHAR(255) | Type of service related to the feedback. |
| date | DATE | Date when feedback was given. |
| age | INT | Age of respondent (optional depending on use). |
| office\_visited | VARCHAR(255) | Office that handled the request. |
| service\_availed | VARCHAR(255) | Service availed by the client. |
| Multiple score fields (q1 to q10) | INT | Scores given for service quality (presumably Likert scale ratings). |
| suggestions | TEXT | Additional suggestions/comments. |
| email | VARCHAR(255) | Email of the feedback giver. |
| created\_at, updated\_at | TIMESTAMP | Timestamps for record creation and update. |

## ***Relationship Descriptions***

| **Relationship** | **Description** |
| --- | --- |
| **Users → Roles** | Many users belong to one role. |
| **Users → Maintenance Requests** | A user can submit many maintenance requests. |
| **Maintenance Requests → Maintenance Types** | Each maintenance request belongs to one maintenance type. |
| **Maintenance Requests → Users** | Staff, Director, and Head verify and approve maintenance requests (foreign keys). |
| **Users → Feedbacks** | A user can submit multiple feedback entries. |
| **Maintenance Requests → Feedbacks** | Feedback is given for a specific maintenance request. |

## ***Additional Notes***

* **Authentication & Session Management:** Auxiliary tables such as sessions, personal\_access\_tokens, password\_reset\_tokens, etc., are generated by the framework (probably Laravel) to handle authentication, password resets, and API token management.
* **System Optimization:** Tables like cache, job\_batches, and failed\_jobs are used for background processing, job queuing, and improving performance. They are **not** part of the core business logic but are critical for system stability.

**V. Error Handling**

**Error Log 1**

[2025-03-04 03:43:53] local.ERROR: Namespace declaration statement has to be the very first statement or after any declare call in the script {"exception":"[object] (Symfony\Component\ErrorHandler\Error\FatalError(code: 0): Namespace declaration statement has to be the very first statement or after any declare call in the script at C:\Users\HP\Desktop\gsosystem\gsobackend\app\Models\User.php:3) [stacktrace] #0 {main} "}

*Analysis:*This error indicates that in User.php, something (like whitespace, a comment, or some other code) exists before the namespace declaration at line 3. In PHP, the namespace declaration must be the very first thing in the file, unless there is a declare() statement before it.

*Suggested solution*: Open User.php and remove any blank spaces, comments, or any other content before the namespace App\Models; line. Make sure the namespace statement is at the very top of the file, right after the PHP opening tag <?php.

**Error Log 2**

[2025-03-04 03:43:54] local.ERROR: Namespace declaration statement has to be the very first statement or after any declare call in the script {"exception":"[object] (Symfony\Component\ErrorHandler\Error\FatalError(code: 0): Namespace declaration statement has to be the very first statement or after any declare call in the script at C:\Users\HP\Desktop\gsosystem\gsobackend\app\Models\User.php:3) [stacktrace] #0 {main} "}

*Analysis:*This is the same error as above, just logged at a different time. The file User.php still has invalid content before the namespace declaration.

*Suggested solution:* Same as earlier — clean the top of the User.php file to ensure no text, whitespace, or comment appears before the namespace declaration.

**Error Log 3**

[2025-03-04 03:44:54] local.ERROR: Namespace declaration statement has to be the very first statement or after any declare call in the script {"exception":"[object] (Symfony\Component\ErrorHandler\Error\FatalError(code: 0): Namespace declaration statement has to be the very first statement or after any declare call in the script at C:\Users\HP\Desktop\gsosystem\gsobackend\app\Models\User.php:3) [stacktrace] #0 {main} "}

*Analysis:*Again, this is a repeated occurrence of the same problem in User.php. Every time the application tries to use this file, it encounters the invalid placement of the namespace.

*Suggested solution:* Correct the file by making sure no whitespace, HTML tag, or comment exists before namespace App\Models;.

**Error Log 4**

[2025-03-04 03:45:55] local.ERROR: Namespace declaration statement has to be the very first statement or after any declare call in the script {"exception":"[object] (Symfony\Component\ErrorHandler\Error\FatalError(code: 0): Namespace declaration statement has to be the very first statement or after any declare call in the script at C:\Users\HP\Desktop\gsosystem\gsobackend\app\Models\User.php:3) [stacktrace] #0 {main} "}

*Analysis:*This is the same namespace placement issue happening again. It will keep repeating until the User.php file is properly cleaned.

*Suggested solution:* As above, remove any characters, whitespaces, or unexpected output before namespace App\Models; inside User.php.

**Error Log 5**

[2025-03-04 03:46:54] local.ERROR: Namespace declaration statement has to be the very first statement or after any declare call in the script {"exception":"[object] (Symfony\Component\ErrorHandler\Error\FatalError(code: 0): Namespace declaration statement has to be the very first statement or after any declare call in the script at C:\Users\HP\Desktop\gsosystem\gsobackend\app\Models\User.php:3) [stacktrace]

*Analysis:*Same recurring error with the namespace position in User.php. The application fails during every attempt to load the file.

*Suggested solution:* Ensure the <?php opening tag is immediately followed by the namespace statement, with absolutely nothing before it.

**VI. Security**

The **GSU Gateway** system is designed with a strong focus on security, ensuring the protection of user data, maintaining system integrity, and preventing unauthorized access. The backend is built using Laravel and follows best practices for authentication, encryption, secure coding, and incident response.

## ***Authentication and Access Control***

* **Login and Authentication:** Users log in by submitting their username and password through a secure POST API. Upon successful authentication, **Laravel Sanctum** issues a **Bearer Token**, which is used to authenticate all future API requests.
* **Route Protection:** All sensitive routes are protected using **Laravel’s auth:sanctum middleware**. Additionally, a **custom role-based middleware** ensures that only users with appropriate roles (e.g., admin, head) can access specific system features such as approving requests.
* **Role-Based Access Control (RBAC):** Access to system functionalities is determined based on user roles, ensuring that users can only perform actions permitted to their assigned role.

## ***Data Protection***

* **Password Security:** Passwords are hashed using **Laravel’s bcrypt** or **argon2** algorithms before being stored in the database. Raw passwords are never saved. Passwords are verified during login using **Hash::check()**.
* **Session and Cookie Encryption:** Sessions and cookies are automatically encrypted by Laravel, preventing unauthorized access to session data.
* **Field Encryption:** Sensitive data fields can be encrypted as needed using **Laravel’s Crypt::encryptString()** function.
* **Secure Transmission:** When deployed online, **HTTPS** (SSL/TLS certificates) is enforced to encrypt all traffic between the client and server, protecting sensitive information from interception during transmission.

## ***Secure Coding Practices***

* **Input Validation:** Every API request validates incoming data using Laravel’s **$request->validate()** function, defending against:
  + SQL Injection
  + Cross-Site Scripting (XSS)
  + Malicious or invalid inputs
* **Mass Assignment Protection:** Laravel’s **$fillable** attribute in Models ensures that only intended fields are mass assignable, further securing the application against injection attacks.
* **Dependency Auditing:** Dependencies are regularly scanned for vulnerabilities using the **composer audit** command, and security patches are applied as necessary during production deployments.

## ***Monitoring and Incident Response***

* **Activity and Error Logging:** Laravel automatically logs errors, login attempts, and failed API requests into **storage/logs/laravel.log**. Additional custom logging is implemented for sensitive actions like password resets or failed login attempts.
* **Attack Detection:** Failed login attempts and suspicious API activity are monitored through Laravel’s built-in events and logs.
* **Incident Response Procedures:** In the event of a security breach, the following steps are executed:
  + Revoke all user tokens using **$user->tokens()->delete()**
  + Force password resets if necessary
  + Investigate the breach through application logs
  + Patch identified vulnerabilities and redeploy the system securely

## ***Compliance and Security Standards***

* **Framework Best Practices:** GSU Gateway follows **Laravel’s secure default practices** for authentication, encryption, and input handling.
* **Data Privacy and Protection:** The system aligns with general data privacy policies and security principles to ensure the protection of user information.

**VII. Conclusion**

The GSU Gateway System was designed to provide an organized, efficient, and secure solution for managing maintenance service requests within the university. Through a carefully structured Entity-Relationship Diagram (ERD) and entity definitions, the system ensures clear data relationships, supporting seamless user interactions across all modules.

The Use Case Diagram highlights the core system functionalities and interactions between different user roles: End Users/Clients, Staff, Heads, and Admins. Each role has defined responsibilities, from submitting requests to approving accounts, ensuring proper flow and authorization throughout the system.

Detailed workflows for service requests, notifications, and account creation were carefully designed to promote transparency, accountability, and user engagement. The transition from a user-driven registration model to an admin-controlled account creation process further strengthens the system’s security and operational control.

Architectural decisions were made based on thorough evaluation, prioritizing performance, scalability, security, and ease of maintenance. Technologies such as React.js, Tailwind CSS, PHP, XAMPP, and deployment platforms like Netlify and Railway were selected to support modern development standards and ensure reliability. For quality assurance, tools like Selenium, Postman, OWASP ZAP, and JMeter enhance system robustness through testing and security assessments.

Error handling mechanisms are integrated at multiple levels, providing meaningful feedback to users and maintaining system stability during unexpected events. Security was a major focus across all stages, employing authentication, role-based access control, encryption, secure coding practices, and real-time monitoring.

Overall, the GSU Gateway System embodies a comprehensive, user-centered platform that enhances service efficiency, improves user experience, strengthens accountability, and upholds the highest standards of security. By strategically choosing appropriate technologies and designing a strong system architecture, the team successfully delivered a solution that is scalable, maintainable, and aligned with the university's operational needs.