**🧭 Introduction**

**SMS (Server Management Suite)** is a powerful, all-in-one server management solution built specifically for institutional networks such as universities, data centers, and large enterprises. It aims to simplify and centralize the management of multiple Linux-based and Windows based servers from a single, intuitive web interface. With SMS, system administrators can remotely monitor server health, configure system settings, manage users, and receive proactive alerts — all without needing to log into each server manually.

The suite is built using modern web technologies and a modular architecture, ensuring high performance and ease of extensibility. Whether your infrastructure includes just a few servers or a large fleet, SMS is designed to scale and adapt to your needs, reducing downtime, increasing operational visibility, and improving overall server efficiency.

**🔑 Key Capabilities (Expanded)**

**✅ Centralized Management**SMS offers a **unified dashboard** where administrators can view, manage, and control all connected servers from one place. This eliminates the need to SSH into each server individually, saving significant time and effort. From basic status info to deep configuration, everything is available in a single portal.

Example: An admin can select any server from the dashboard, view its status, execute commands, manage users, or restart services — all remotely and securely.

**⏱ Real-time Monitoring**

SMS continuously collects live metrics from all connected servers, displaying data like:

* **CPU Usage**: Detect processing spikes or bottlenecks
* **Memory Usage**: Monitor available and used RAM
* **Disk Usage**: Track storage space across partitions
* **Network I/O**: Analyze incoming and outgoing traffic

These metrics are updated in real-time (every 30 seconds), allowing admins to respond quickly to system stress or anomalies. Visual graphs and historical data further help in trend analysis and capacity planning.

**⚙️ Configuration Management**

Administrators can perform critical system and network configurations directly from the SMS dashboard:

* Edit **hostname**, **timezone**, or **system credentials**
* Manage **network interfaces**, **routes**, and **firewall rules**
* Apply **SSH keys** , Password Management

This remote configuration capability ensures consistent setups across servers and allows quick response to any operational requirements.

**🚨 Alert System**

The built-in alert system ensures administrators are always informed when something goes wrong. It continuously checks critical resource metrics and generates alerts when thresholds are breached:

* CPU usage above 90%
* Memory usage above 95%
* Disk nearly full
* Network issues like packet loss or downtime

Each alert includes severity levels (Info, Warning, Critical), timestamps, and the affected server. Alerts can be filtered, marked as seen/unseen, and deleted (admin-only). This enables timely interventions and proactive maintenance.

**🔐 Role-based Access Control**

To ensure secure and controlled access, SMS provides two main roles:

* **Admin**: Full access to all features, including configuration, user management, logs, and alerts.
* **Viewer**: Read-only access, suitable for monitoring and status viewing without the ability to make changes.

Role-based UI customization ensures each user sees only the features relevant to their role, reducing complexity and potential misuse.

**🔒 Secure Communication**

Security is a core principle in SMS. All communication between the frontend, backend, and client agents is protected using:

* **JWT (JSON Web Tokens)** for secure user authentication
* **Access tokens** for identifying and managing client servers
* **Input validation and sanitization** to prevent malicious commands
* **Encrypted communication protocols** for production deployments (via HTTPS)

These measures ensure that only authorized users and devices can interact with the system and that data remains safe and tamper-proof.

**🔧 Installation Guide – Detailed Walkthrough**

To successfully set up and run the SMS system, you’ll need to install and configure three components:

1. **Backend** – Handles logic, APIs, database interaction, client communication
2. **Frontend** – The React-based dashboard for user interaction
3. **Client (Agent)** – A Go executable deployed on target servers to collect metrics and perform tasks

**✅ Prerequisites (System Requirements)**

Before installation, make sure you have the following installed:

| **Tool/Software** | **Version** | **Purpose** |
| --- | --- | --- |
| **Go** | 1.19+ | For building backend server & client agent |
| **Node.js** | 16+ | For frontend development (React, Vite, etc.) |
| **PostgreSQL** | 12+ | To store users, server info, metrics, logs |
| **Linux Servers** | Any distro with sudo | For running the client with elevated permissions |

**🛠 Backend Setup (Go + PostgreSQL)**

**🔹 Step 1: Clone the Repository**

bash

CopyEdit

git clone <repository-url>

cd backend

This pulls the backend source code into your system and moves into the backend directory.

**🔹 Step 2: Install Dependencies**

bash

CopyEdit

go mod download

This installs all required Go packages (modules) used in the project like JWT, PostgreSQL driver, etc.

**🔹 Step 3: Environment Setup**

bash

CopyEdit

cp .env.example .env

This creates a .env file with environment variables like:

* Server port
* Database URL
* JWT secret
* Client Port

**Edit the .env** file to set your actual PostgreSQL credentials and server details.

Example:

env

CopyEdit  
  
CLIENT\_PORT=2210

SERVER\_PORT=8000

DATABASE\_URL=postgres://admin:yourpassword@localhost:5432/sms

JWT\_SECRET=mySuperSecretKey

**🔹 Step 4: Initialize the Database**

bash

CopyEdit

go run temp/dbinit.go

This script does the following:

* Connects to the PostgreSQL database
* Creates necessary tables (users, alerts, server\_devices, etc.)
* Sets up initial schema and relationships

It ensures that your backend is ready to store and retrieve data.

**🔹 Step 5: Run the Backend Server**

bash

CopyEdit

go run main.go

Starts the backend API server on the configured port (e.g., 8000). It’s now ready to:

* Accept frontend API calls
* Communicate with registered client agents
* Store data into PostgreSQL

**💻 Frontend Setup (React + Vite)**

**🔹 Step 1: Move to Frontend Directory**

bash

CopyEdit

cd frontend

**🔹 Step 2: Install Dependencies**

bash

CopyEdit

npm install

Installs React, React Router, Vite, and other frontend packages.

**🔹 Step 3: Environment Configuration**

bash

CopyEdit

cp .env.example .env

Edit the .env to point to your backend server:

env

CopyEdit

VITE\_BACKEND\_URL=http://localhost:8000

This allows the frontend to talk to the backend via REST APIs.

**🔹 Step 4: Run in Development Mode**

bash

CopyEdit

npm run dev

Starts the frontend locally (usually on http://localhost:3000) with hot-reloading. Great for testing and development.

**🔹 Step 5: Build for Production**

bash

CopyEdit

npm run build

Compiles the code into optimized, static files (in /dist folder) for production deployment.

**🖥 Client Deployment (Server Agent)**

The **client** is a Go executable that runs on each target Linux server that you want to manage.

**🔹 Step 1: Place the Executable**

Copy the client-executable file (compiled from Go) to your target server.

**🔹 Step 2: Run with Sudo Privileges**

bash

CopyEdit

sudo ./client-executable --port 2210

* Runs the agent on port 2210
* Sudo is required to access system-level resources (CPU, memory, logs, etc.)

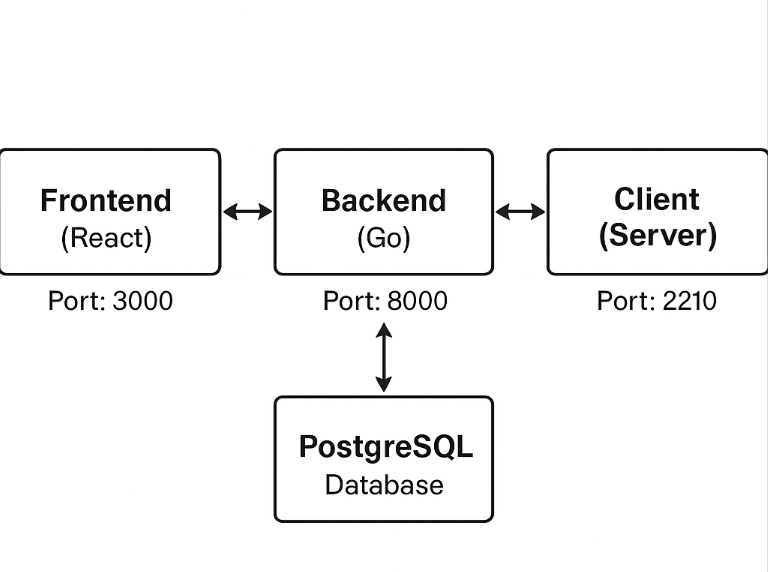
**🔹 How It Works**

* On startup, the admin need to enter the access token given by the Frontend to the client this is need to done for secure commuication.
* It **stays connected**, waiting for requests from the backend (e.g., "Give me the disk usage", "Change hostname").
* It **collects real-time metrics** (every 30 sec) and sends them to the backend.
* The backend then updates the dashboard for monitoring.

**Architecture Overview**

**📐 System Architecture**

yaml



The SMS platform is based on a **modular architecture** where the frontend, backend, and client agents interact seamlessly to provide a real-time, secure, and centralized server management system.

**🔄 Communication Flow**

* **Frontend → Backend**: Users interact with the system through a web interface built using React. Actions like login, selecting a server, or executing a command are sent as HTTP REST API requests to the backend.
* **Backend → Database**: The backend uses SQLC with PostgreSQL to store and retrieve critical data like server configurations, metrics, user sessions, alert histories and access token.
* **Backend → Client**: When a user performs an action (e.g., run a command or check disk usage), the backend securely forwards this to the appropriate client agent using the registered access token and server ID.
* **Client → Backend**: Clients, running on each managed server, collect metrics and handle operations. The results (e.g., CPU usage, command output) are sent back to the backend.
* **Backend → Frontend**: After receiving responses from the client, the backend forwards the processed data to the frontend in real time, ensuring the dashboard remains responsive and informative.

**🚀 Deployment Strategy**

* **Production Environment**: The backend and frontend are deployed on a central server within the institution’s internal network, minimizing exposure to external threats.
* **Network Access**: All operations occur within a **secured local network**, ensuring sensitive system commands and logs never traverse the public internet.
* **Client Distribution**: The client-executable is copied to each Linux server to be managed. It runs with sudo to access low-level system resources and connects to the central backend securely using JWT tokens.

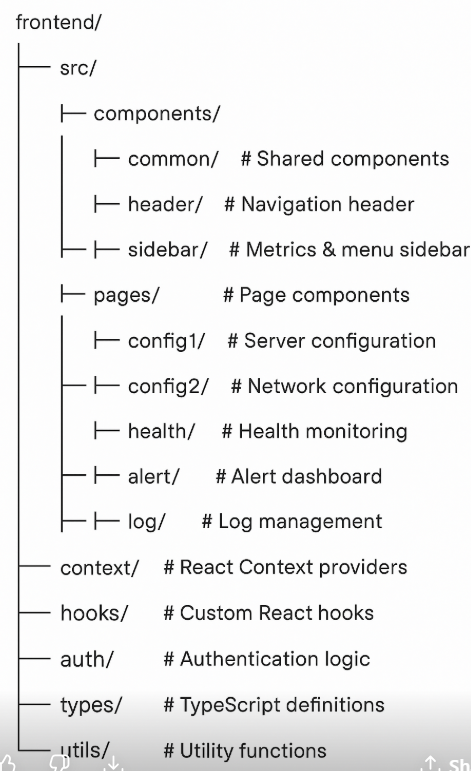
**💻 Frontend**

**🔧 Technology Stack**

* **React 18+** with **TypeScript** for component-based UI development and type safety.
* **React Router DOM** for seamless page navigation across dashboard sections.
* **Context API** for sharing state like active server, user roles, and authentication across components.
* **Custom CSS** styled with a clean, professional **dark theme**.
* **React Icons** used to provide a user-friendly, visually intuitive interface.

**📁 Project Structure**

Bash



**🌟 Key Features**

* **Responsive Design**: Optimized for both desktops and mobile browsers, making it easy to monitor servers on the go.
* **Live Monitoring**: Real-time streaming of CPU, memory, disk, and network usage graphs for all registered servers.
* **Device Management**: Easily switch between registered servers to monitor or manage specific ones.
* **Role-based Interface**: Viewers have limited access; admins can configure servers, users, and alerts.
* **Dark Mode UI**: Clean dark interface with intuitive layout and minimal visual clutter.

**🔁 State Management**

* AppContext: Central state store for global variables like selected device, sidebar status, etc.
* AuthContext: Manages JWT tokens, login sessions, and role-based access control.
* HealthContext: Holds periodic health metric updates for each server.

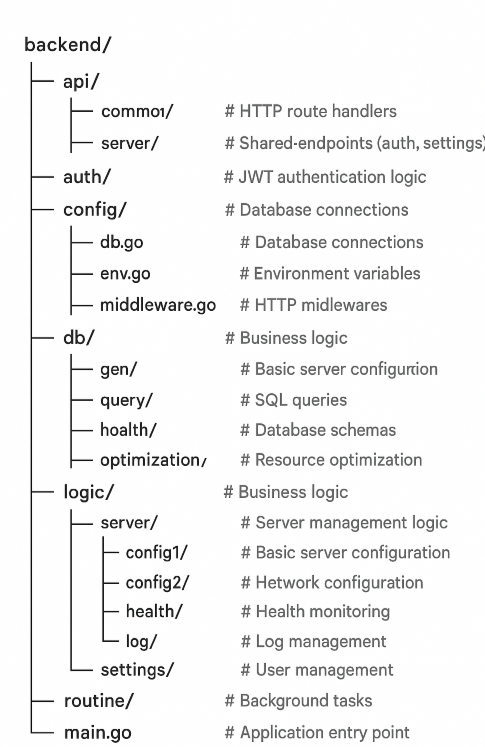
**🔧 Backend**

**🔧 Technology Stack**

* **Go 1.19+** for performance, concurrency, and native networking capabilities.
* **PostgreSQL** for persistent data storage and relational queries.
* **SQLC** for generating safe, fast, type-checked Go code from SQL.
* **JWT** for secure, stateless user authentication and role enforcement.
* **.env config** for flexible deployment with environment-specific variables.

**📁 Project Structure**

Graphql



**🛠 API Architecture**

* **Public Routes (/api/auth/)**: Handle login, logout, token refresh.
* **Protected Routes (/api/server/)**: Used for managing server operations, visible only to logged-in users.
* **Admin Routes (/api/admin/)**: Allow user creation, server config, and system changes — restricted to admin users.

**🧰 Middleware Stack**

* **Security Headers**: Add headers like X-Content-Type-Options, X-XSS-Protection to prevent attacks.
* **CORS Handling**: Allows frontend hosted on different port/domain to communicate with backend securely.
* **JWT Authentication**: Validates incoming requests with Bearer tokens.
* **Role Authorization**: Ensures restricted routes can only be accessed by users with proper roles.

**🗃 Database Design**

**General Tables:**

* users: Stores login credentials and roles (admin/viewer).
* sessions: Tracks active login sessions.
* mac\_access\_status: Controls which devices have access enabled.

**Server Tables:S**

* server\_devices: List of all registered client machines with IDs and tokens.
* alerts: Stores alert logs for high resource usage, failures, etc., with seen/unseen statuses.

**🖥 Client / Server Controller**

**🧩 Deployment**

* The client is deployed as a lightweight Go binary on each server to be managed.
* Runs on **port 2210**, consistent across all installations.
* Uses a unique access token and device ID to register itself securely with the backend.

**🔍 Functionality**

* **Monitoring**: Reads system data (CPU, RAM, etc.) from /proc and sysfs to report to backend.
* **Command Execution**: Executes only whitelisted or validated system commands.
* **Network Configuration**: Modify IPs, gateways, DNS settings remotely.
* **Service Management**: Start, stop, and check the status of system services like nginx, sshd.
* **Log Fetching**: Uses journalctl and system logs to fetch data based on request filters.

**🔐 Security**

* **JWT-secured communication**: Ensures only authenticated and authorized requests are executed.
* **Command validation**: Prevents shell injection or misuse of critical system commands.
* **Role-based request execution**: Ensures commands sent to client are only initiated by authorized admin
* **🛠 Tech Stack**
* The Server Management Suite (SMS) is built using a modern and efficient technology stack that ensures performance, scalability, and maintainability. The system leverages well-established tools for both frontend and backend development along with essential tools for smooth DevOps and development workflows.
* **🎨 Frontend Technologies**

| **Technology** | **Purpose** | **Version** |
| --- | --- | --- |
| **React** | Core UI framework for building interactive, component-based user interfaces. Enables fast rendering and modular development. | 18+ |
| **TypeScript** | Superset of JavaScript that introduces static typing, making the codebase safer, easier to refactor, and less prone to bugs. | 4+ |
| **React Router** | Enables client-side routing and navigation between multiple pages or views without reloading the page. | 6+ |
| **React Icons** | Provides a library of popular icons (e.g., FontAwesome, Feather, etc.) to create visually appealing UI components. | 4+ |
| **Vite** | A next-generation front-end tooling and build system optimized for speed, hot module reloading, and better development experience. | 4+ |

* These technologies collectively power the responsive, fast, and user-friendly dashboard UI of the SMS system.
* **⚙️ Backend Technologies**

| **Technology** | **Purpose** | **Version** |
| --- | --- | --- |
| **Go (Golang)** | A high-performance, statically typed programming language used for building the backend server logic and concurrent client handling. | 1.19+ |
| **PostgreSQL** | A powerful, open-source relational database system used to store user data, server metadata, logs, alerts, and configurations. | 12+ |
| **SQLC** | Generates type-safe Go code from SQL queries, allowing precise control over data access and reducing bugs in database operations. | 1.18+ |
| **JWT** | JSON Web Token used for secure and stateless authentication. Ensures that only authorized users can access or control resources. | - |
| **godotenv** | Loads environment variables from .env files during local development for configuration flexibility and security. | - |

* These backend technologies ensure secure, reliable, and efficient server-side operations, enabling real-time communication between clients and the central backend.
* **🧰 Development Tools**

| **Tool** | **Purpose** |
| --- | --- |
| **Git** | Version control system used for managing code repositories, collaboration, and tracking changes. |
| **VS Code** | A lightweight yet powerful source code editor with support for Go, TypeScript, and integrated terminal/debugging. |
| **Postman** | A popular tool for testing and debugging REST APIs. Helps verify endpoint functionality during development. |

* These tools are essential in the development lifecycle — from writing and testing code to managing infrastructure and ensuring reliable deployments.

**📊 Modules & Features**

The SMS platform is modular and scalable, offering distinct functional areas that streamline server and network management. Each module is tailored for real-time insights, efficient operations, and secure administrative control.

**1. Configuration Management (Config1)**

**📌 Purpose:** To enable foundational control over server systems from a centralized interface.

**🔧 Features:**

* **Device Registration:** Securely add or remove Linux servers from the centralized dashboard using access tokens.
* **Basic Info Management:** View and update server details like hostname, operating system info, timezone, etc.
* **Command Execution:** Execute safe and validated shell commands remotely via a terminal-style interface.
* **Password Management:** Change or reset user account passwords on remote servers with audit logging.
* **SSH Key Management:** Upload, list, and delete authorized SSH public keys to enable passwordless access.
* **Server Overview:** Displays live information about the server's health, uptime, and system summary (OS, CPU, RAM).

**2. Network Configuration (Config2)**

**📌 Purpose:** Manage and fine-tune server networking to ensure availability and security.

**🔧 Features:**

* **Network Basics:** Shows current IPs, subnet masks, MAC addresses, and interface names.
* **Interface Management:** Enable/disable specific interfaces such as eth0, wlan0, etc., without rebooting.
* **Routing Table Management:** View existing routing entries and add/delete static routes with gateway validation.
* **Firewall Management:** Secure the server by configuring rules using iptables or nftables, with real-time status.
* **Service Restart:** Restart or reload network services like NetworkManager or systemd-networkd remotely.

**3. Health Monitoring**

**📌 Purpose:** Provide real-time metrics and trends for server resource usage and performance.

**📈 Metrics Monitored:**

* **CPU Usage:** Percent of processor time in use by system/user processes.
* **RAM Usage:** Current memory load and availability in MB/GB and percentage.
* **Disk Usage:** Partition-specific storage consumption with thresholds.
* **Network I/O:** Total incoming/outgoing data, packet rate, and bandwidth usage.

**🔧 Features:**

* Real-time updates pushed every **30 seconds**.
* Graphs and visualizations of historical trends (hour/day/week).
* Color-coded indicators (green/yellow/red) for quick status recognition.
* Compare health metrics across multiple registered servers.

**4. Alert System**

**📌 Purpose:** Automatic detection and notification of system anomalies to reduce downtime.

**⚠ Alert Triggers:**

* **CPU Overload:** Sustained usage > 90%.
* **Memory Saturation:** RAM usage exceeding 95%.
* **Disk Space Full:** Storage capacity over 95%.
* **Network Instability:** Loss of connectivity or excessive latency.

**🔧 Alert Management:**

* **Status Tracking:** Mark alerts as seen or unseen, with timestamps.
* **Severity Levels:** Classify alerts as Info, Warning, or Critical for priority-based responses.
* **Admin Actions:** Only admins can delete or resolve alerts.
* **Advanced Filtering:** Sort alerts by server, type, or severity level.

**5. Log Management**

**📌 Purpose:** Aggregated view of logs from managed servers for diagnostics and audit trails.

**🔧 Features:**

* **Real-time Streaming:** View logs as they are generated using journalctl.
* **Flexible Filtering:** Filter logs by date, time, log level, or specific systemd services.
* **Multi-Server Log Access:** Switch between servers without logging out.
* **Full-text Search:** Keyword-based searching to trace issues quickly.

**6. Resource Optimization**

**📌 Purpose:** Tools and suggestions to enhance system performance and cleanup resources.

**🔧 Features:**

* **Service Management:** List active services and restart misbehaving ones.
* **Process Monitoring:** View all running processes with CPU/memory consumption.
* **Cleanup Utilities:** Clear unused files, old logs, and temp directories.
* **Performance Tuning:** Suggestions based on system usage patterns (e.g., swap optimization, service disabling).

**7. User Management (Settings)**

**📌 Purpose:** Handle authentication, authorization, and user settings in a secure way.

**🔧 Features:**

* **User Creation:** Add new users with Admin or Viewer roles.
* **Role Assignment:** Define access levels and restrict UI visibility based on roles.
* **User Deletion:** Safely remove old or inactive accounts with audit logs.
* **Profile Updates:** Let users update their own email, password, and display name.

**⚙ Configuration**

Configuration settings are managed via environment variables and .env files for flexibility and deployment ease.

**🌐 Environment Variables (.env)**

bash

CopyEdit

# Server Configuration

SERVER\_PORT=8000 # Backend service port

CLIENT\_PORT=2210 # Fixed port for all client agents

CLIENT\_PROTOCOL=http # Protocol for backend-client communication

# Database Configuration

DATABASE\_URL=postgres://user:password@localhost:5432/snsms

# PostgreSQL connection string

# Security

JWT\_SECRET=your-super-secret-jwt-key

# Secret key for token signing and verification

# Logging

LOG\_LEVEL=info # Logging level: debug/info/warning/error

These variables control essential behavior like port bindings, database connections, and security handling. Make sure to keep .env files secure and excluded from version control.

**🗃 Database Initialization**

The database is initialized using a Go-based script that automatically creates necessary tables and schemas:

bash

CopyEdit

go run temp/dbinit.go

This step ensures all essential database structures (users, sessions, server\_devices, alerts, etc.) are ready for operation.

**🖥 Client Configuration**

The client agent runs as a background service on each managed Linux server.

* **Port:** Listens on **port 2210**, consistent across all installations.
* **Permissions:** Requires sudo privileges to access system-level data and services.
* **Auto-registration:** On startup, each client authenticates and registers with the backend using a **pre-shared access token**.
* **Token-Based Communication:** All communication between backend and client is **JWT secured** to prevent tampering and unauthorized actions.

**🔐 Security**

Security is a critical pillar of the Server Management Suite (SMS). From session control to secure command execution and encrypted communication, SMS incorporates multi-layered protection mechanisms across every component.

**Authentication & Authorization**

* **JWT Tokens:**  
  SMS uses industry-standard **JSON Web Tokens (JWT)** for user authentication. Upon login, the backend generates a signed token that stores user identity and role.
* **Role-based Access:**  
  Users are assigned roles:
  + **Admin:** Full control over configuration, monitoring, user management
  + **Viewer:** Read-only access to metrics and logs  
    Route protection ensures only authorized users access privileged endpoints.
* **Token Refresh:**  
  Refresh tokens allow automatic renewal of expired JWTs, ensuring long sessions without forcing users to re-login frequently.
* **Session Management:**  
  All user sessions are securely stored in the sessions table. Logout operations clean up the session both on client and server, invalidating tokens.

**Communication Security**

* **Bearer Tokens:**  
  Every request from frontend and client includes a JWT token in the Authorization: Bearer <token> header. The backend validates it before any operation.
* **Input Validation:**  
  All user-provided input is validated and sanitized to prevent injection attacks (e.g., command injection, XSS).
* **Command Validation:**  
  Commands executed by the client agent are filtered through a command whitelist or validator to block malicious or dangerous operations.
* **Access Control (Clients):**  
  Each client/server controller registers using a unique, **device-specific access token**. This ensures only known servers can connect.

**Network Security**

* **Internal Network Deployment:**  
  Backend and clients are deployed within a secure **institutional LAN** environment to reduce exposure to external threats.
* **CORS Protection:**  
  Backend is configured to accept requests only from known frontend origins to prevent Cross-Origin Resource Sharing attacks.
* **Security Headers:**  
  HTTP responses include:
  + X-Content-Type-Options: nosniff
  + X-XSS-Protection: 1; mode=block
  + Content-Security-Policy: default-src 'self'
* **HTTPS Ready:**  
  The system is configured for **SSL/TLS** termination. Production deployment uses Nginx or Caddy for HTTPS support via Let's Encrypt or self-signed certificates.

**Data Protection**

* **Password Hashing:**  
  Passwords are hashed using **bcrypt** before storing in the database, making it infeasible to retrieve even in case of DB compromise.
* **Token Encryption:**  
  While JWTs are signed (not encrypted) by default, SMS uses **strong secret keys** and short expiry times. Optional encryption can be added for sensitive claims.
* **Database Security:**  
  All queries are handled through **SQLC** generated code with parameterized statements to eliminate SQL injection risks.
* **Audit Logging:**  
  Sensitive operations (e.g., login, command execution, password changes, user management) are logged for traceability and accountability.

**🎯 Conclusion**

The **Server Management Suite (SMS)** provides a powerful, centralized platform for managing Linux servers in institutional and enterprise environments. By combining modern web technologies with a secure and scalable backend, SMS offers real-time monitoring, streamlined configuration, robust user access control, and automated alerting — all within an intuitive interface.

The suite's modular design ensures each component (e.g., configuration, health monitoring, alerting, user management) operates independently yet integrates seamlessly. This makes the system both flexible and extensible, allowing easy addition of new features like backup management or compliance auditing.

Security remains a top priority with features like JWT-based authentication, role-based access, input sanitization, and encrypted communications. Deployment is straightforward across clients and servers, with built-in tools for database initialization, client auto-registration, and system health tracking.

Overall, SMS is built to simplify complex server administration tasks while maintaining high standards of reliability, visibility, and control — making it a practical solution for both small teams and large-scale IT infrastructures.