**SmartGate Web App Documentation**

**Project Contributers**  
Developed by *Talha Ahmed, Sabin Sapkota, and Christopher Lafferty*  
Designed and Planning by *Dharsha Jaikumar*

**Tech Stack**  
Python, JavaScript, HTML, CSS, Jinja2 templating, and SQL.

Features

1. Dashboard: View scrolling images and a quote reflecting SmartGate’s mission.
2. Login: Sign in with a dummy account or via GitHub.
3. Alerts Page: View static alerts fetched from the database (no real-time popups).
4. Gates Page: See gate images, numbers, and current status.
5. Admin Control: Admins can open/close gates using buttons.
6. User Overview: See live user data updated in real-time.
7. Role Management: Any user can promote/demote others between user and admin.
8. User Removal: Any user can remove others, instantly logging them out.
9. About Page: Read static information about the app.
10. Stats Page: View hardcoded data about gate openings, gate closes, detected animals and their stats shown in a table and histogram chart

Planned Features

1. Real-time Animal Data View: Live updates showing data about what animals are detected
2. Improved Permission Checks: Clearly define and enforce what regular users and admins can do.
3. Live Statistical Charts: Interactive graphs of animal detection
4. Alert Notifications: Instant in-app alerts for events related to the gates and users
5. User Permission Management: Control over user access levels
6. Gate Live Feed: Show live camera feeds from the gates on the gates page
7. Fetch Live Data Gate Usage & Detection: View real time data on the stats page about gate openings, gate closes and detected activity
8. Mobile-Friendly Interface: Optimized design for smartphones and tablets
9. Email or SMS Alerts: Send external alerts for critical gate or system events.
10. Scheduled Gate Control: Automatically open/close gates at set times

Note: All the features and potential features will be added to the remake of the SmartGate web application.

1. **Frontend Implementations**

* Layout Page (layout.html)
  + Template Base: Uses Jinja2 blocks for reusable layout.
  + Header & Nav: Shows logo and links to key pages.
  + User Section: Displays login or user info based on session.
  + Buttons: Login via a github account and a dummy login button for testing; switches to logout/profile when logged in.
  + Styling: Loads styles from Tabulator and custom CSS from site.css.
    - Tabulator Styles is for the tables in stats.html
    - site.css
      * Provides a layout and style for UI elements across pages that extend the layout.html.
  + Page Context: Sets data-page for scripts to use.
  + Footer & JS: Adds footer and loads site scripts such as websocket-client.js and site.js
    - websocket-client.js:
      * WebSocket Setup: Opens a connection if not already established
      * Event Handlers: Processes events like user updates, redirects, and alerts
      * Session Handling: Sends session username and handles specific page data
    - site.js
      * Linked to Layout: This script is linked to layout.html but it is not heavily used.
* Dashboard (index.html)
* Extends Layout: Inherits base HTML from layout.html.
* Page Title: Sets title block to "Dashboard".
* Video Background: Plays looping wildlife video behind content.
* Image Slider: Rotates wildlife images in a fading carousel.
* Description Text: Shows SmartGate mission statement.
* JS Slider Logic: Internal Script auto-changes image set every 3 seconds.
* Gates (gates.html)
  + Extends Layout: Inherits structure from layout.html.
  + Sets Title: Page title is set to “Gates”.
  + Background Video: Adds a looping wildlife-themed video.
  + Overlay Layer: Darkens video for better content visibility.
  + Page Heading: Displays “Gates Management” as the main title.
  + Gate Cards: Six gate sections show camera feeds as static images and info.
  + Gate Status: Shows each gate's status as “Closed” (default).
  + Gate Controls: Buttons allow users to open/close gates.
  + Grid Layout: Feeds arranged in a responsive 3x2 grid.
  + Custom JS + CSS: Uses gate.js and inline styles for logic and design.
    - Initial Status Push: On page load, each gate's status is sent to the backend for record keeping.
    - Permission-Based Actions: Users must have proper permissions (via checkUserPermission) to open or close gates.
    - Live Status Updates: Button clicks trigger real-time status changes and notify the backend (/update\_gate\_data).
* Users (users.html)
  + User Table: Lists usernames, roles, status, and a remove button.
  + Live Updates: Uses WebSocket to update the table in real time.
  + Remove User: Sends a request to delete a user and removes the row.
  + Role Modal: Clicking a row opens a modal to change the user's role.
  + Background Video: Adds a looping video for visual appeal.
  + Status Colors: Green for logged in, red for logged out.
  + Internal Script:
    - User Table: Displays Users with roles and status allowing role changes.
    - Actions: Supports user removal and role updates via WebSocket.
  + Internal Styling:
    - background: Fullscreen video with dark overlay.
    - Table: Styled with padding, hover effects, and status colors.
    - Modal: Centered box with dropdown and action buttons.
* Stats (stats.html)
  + Background Video
  + Contains three tables
    - Gates (gate\_status, gate\_no\_opens, gate\_no\_closes)
    - Animals (animal\_id, animal\_name, is\_endangered (true/false), is\_threat (true/false), time\_of\_detection aka time\_stamp)
    - Stats (total\_animals, total\_threats, total\_endangered, % threats, % endangered, % both)
    - Histogram (Animal Name, Animal Count)
  + Internal CSS and JavaScript
    - JavaScript
      * Data Contained under <script></script>
      * Using API [PlotlyJS - > Show Histogram Chat & Tabulator JS - > Show tables]
      * Styling for the table rows
    - CSS
* Arranging the layout of the tables and histogram
* Added Some Spacing
* Alerts Page (alerts.html)
  + Background Video
  + Alerts table (data gathered from DB) with the following attributes:
    - Select Checkbox
    - Alert No
    - Description
    - Level
    - Date and time
  + Internal CSS and JavaScript
    - CSS:
      * Provides a pleasant visual appearance
    - JavaScript:
      * Alert Table with Actions: Displays alerts from db.
* About Us (about.html)
  + Background Video
  + Team Intro: Shows who the team is
  + Solutions: Lists key platform features
  + Mission & Vision: Goals shown side by side
  + Features List: Quick highlights with icons
  + Internal CSS:
    - Provides a way to organize the sections of the about page neatly

1. **Backend Implementations**

* Controllers
  + db\_controller.py
    - Database Connection
      * Loads DATABASE\_URL from Environment
      * get\_db\_connection(): Establishes a new connection to the PostgreSQL DB using psycopg
    - Connection Health Check
      * check\_db\_connection(): Verifies that the database connection works by running a SELECT 1 query.
    - User
      * insert\_user(user): Adds a user to the DB if they don’t already exist (checks by user\_id or username).
      * remove\_user(username): Deletes a user from the DB by username.
      * change\_role(username, role\_name): Updates a user's role based on role name.
      * mark\_user\_logged\_in(username): Sets user's status to "Logged in" in user\_overview.
      * mark\_user\_logged\_out(username): Sets user's status to "Logged out" in user\_overview.
      * is\_user\_logged\_in(username): Returns True if user is marked as logged in in user\_overview.
      * clear\_all\_users(): Deletes all users from the users table.
    - Permission Handling:
      * check\_permission(username, perm\_name): Checks if a user has a specific permission (stored as JSONB).
      * add\_permission(username, perm\_name): Adds a permission to a user’s permission set.
      * remove\_permission(username, perm\_name): Removes a specific permission from the user.
    - User Overview
      * get\_user\_overview(): Retrieves all entries from the user\_overview table.
    - Alert System:
      * get\_all\_alerts(): Fetches all alerts from the alerts table.
      * add\_alert(alert\_desc, alert\_level): Inserts a new alert with a description and severity level.
      * delete\_alert(alert\_no): Removes an alert by its number.
    - Gate Management:
      * add\_gate(gate\_no, gate\_status): Adds a new gate to the gates table.
      * update\_gate\_status(gate\_no, new\_status): Updates gate status and increments open/close counters if changed.
  + main\_controller.py
    - Basic page routing (Dashboard, Gates, Alerts, Stats, About Us, Users)
    - Login via GitHub OAuth and Dummy login
    - Session management and session-based username fetching
    - WebSocket setup for live data push
    - WebSocket event dispatcher (using event\_registry)
    - WebSocket initial alert data push on connect
    - WebSocket dynamic event handler with await handler(websocket, data)
    - WebSocket cleanup on disconnect
    - WebSocket state tracking with websocket\_state
    - Remove user safely with self-removal prevention
    - Kick out removed users via WebSocket redirect and closure
    - Health check endpoint (/health) with DB connectivity check
    - Fetch user overview data for broadcast
    - Fetch alert data for broadcast
    - Broadcast updated user data (broadcast\_user\_overview())
    - Broadcast updated alert data (broadcast\_alert\_data())
    - broadcast\_data(event, data) sends to all connected WebSocket clients
    - Helper function to get user from session (get\_user\_from\_session())
    - Helper functions to fetch alert and user data from DB
    - Generic render\_page() utility for templated responses
    - GateData and updateGateData Pydantic models for input validation
    - Add gate data via POST endpoint (/add\_gate\_data)
    - Update gate status via POST endpoint (/update\_gate\_data)
    - GitHub OAuth token exchange and user info fetch
    - User DB entry and login marking after successful OAuth
    - Dummy user insertion and login tracking
    - Logout clears session and broadcasts update
    - Redirect users who are removed (/removed)
    - Dependency-based username retrieval (/get-username)
    - Check permission API (/check-permission) using username and permission
    - HTML templating via Jinja2 with context injection
    - Cleanup of disconnected WebSocket clients in broadcast\_data()
    - Tracking WebSocket connections and user association in websocket\_state
    - Optional decorator (commented out) for automatic user overview broadcast
  + websocket\_events.py
* Registers WebSocket events dynamically using event\_registry
* Uses @register\_event decorator to bind handlers to event names
* Handles client initialization via "init" event and updates websocket\_state
* Sends live user data via "user\_overview" event using fetch\_user\_data()
* Allows role changes via "change\_role" event and broadcasts updates with broadcast\_user\_overview()
* Sends real-time alert data via "alert\_data" event using fetch\_alerts\_data()
* Avoids circular imports with localized imports inside event handlers
* Maintains active WebSocket client metadata in websocket\_state
* Models
  + user\_schema.sql
* roles table defines available roles like 'user' and 'admin' with auto-incremented IDs
* Inserts default roles into the roles table
* users table stores users linked to a role via foreign key
* perms table defines available permissions like view or control actions
* Inserts permissions such as 'view\_gate', 'open\_gate', etc., into perms
* role\_permissions table maps roles to permissions with uniqueness enforced
* Assigns various permissions to roles, including full control to admins
* user\_perms table stores a JSONB list of permissions per user
* refresh\_user\_perms() updates user permissions based on current role
* on\_user\_update() trigger function calls refresh after insert or role change
* Triggers call on\_user\_update() after user insert or role change
* on\_user\_delete() removes user permissions on user deletion
* Trigger deletes permissions from user\_perms after user removal
* user\_status\_enum defines 'Logged in' and 'Logged out' states
* user\_overview table tracks username, role, and login status
* refresh\_user\_overview\_on\_insert() adds user to overview as 'Logged in’
* Trigger adds to user\_overview after new user insert
* remove\_from\_user\_overview\_on\_delete() removes user from overview
* Trigger removes from user\_overview on user deletion
* update\_user\_overview\_role() updates role in overview on change
* Trigger updates overview role info when user's role\_id is modified
  + gate\_schema.sql
* detected\_animals table stores detected animals with type, name, threat, endangered status, and timestamp
* Mock data inserted for various animals including endangered and non-threat types
* alert\_level\_enum enum defines alert levels: info, warning, and critical
* alerts table records system alerts with description, level, and timestamp
* Sample alert entries inserted for testing system behaviors
* gate\_status\_enum enum defines gate states as Open or Closed
* gates table tracks each gate’s status along with open/close counters
* animal\_name\_vs\_count table stores count of each detected animal name
* upsert\_animal\_name\_vs\_count() updates the count table from detected animal entries
* update\_animal\_count\_trigger automatically runs upsert after new animal detection
* var\_stats table stores summary stats like totals and percentages
* refresh\_var\_stats() recalculates stats on threats, endangered, and combined categories
* refresh\_var\_stats\_after\_insert trigger updates var\_stats after inserting a new detection

1. **Docker Composition**

* sgwebimage
* Builds the SmartGate web app from the local directory.
* Exposes the app on port 8000.
* Connects to the Postgres database using environment variables.
* Depends on postgres service to start first.
* Mounts project (web-app) files inside the docker container.
* Joins the custom network
* Configured a health check to monitor app availability at /health.
* Postgres
* Runs a PostgreSQL 13 database container.
* Sets up database smartgatedb with user (admin) and password (smartgate).
* Mounts SQL schema files to initialize tables automatically on startup.

1. **Future Implementations**

The next generation of the SmartGate web application will be a complete remake using modern JavaScript technologies. The new version will emphasize speed, scalability, and maintainability, leveraging Node.js and associated tooling to streamline development.

* Key Goals
  + Pure JavaScript Frontend & Backend: Replace the current Python-Jinja2 stack with a full JavaScript solution using Node.js.
  + Microservice Architecture: Use FastAPI (Python) as a dedicated service for gate control and hardware communication.
  + Improved Real-time Communication: Enhance live data updates using Socket.IO on the Node.js side and WebSockets via FastAPI for gate integration.
  + Modular Codebase: Keep services loosely coupled and easily testable.
* Planned Stack & Tooling
  + Node.js + Fastify: Lightweight and fast backend server for handling HTTP routes and serving static files.
  + Fastify Plugins: Including @fastify/static for serving static content and others for request validation, cookie/session handling, etc.
  + Socket.IO: For real-time communication with the frontend (user data, gate alerts, live updates).
  + FastAPI (Python): Acts as a hardware-facing microservice, communicating with physical gates and broadcasting state changes.
  + PostgreSQL: Retain current PostgreSQL schema for storing users, gate data, alerts, and statistics.
  + Dockerized Development: The new app will be bundled in a Docker container with a Node.js-based entrypoint, ready for deployment.

Repository Link  
The work on the new implementation will be hosted at:  
<https://github.com/TheOpenSI/SmartGate/tree/webapp/src/web-app-js>What’s Coming  
All current features and the planned improvements listed above — including mobile optimization, scheduled gate operations, external notifications (email/SMS), permission refinements, and real-time dashboards — will be integrated into this remake.