**Web UI Automation Manual**

**Table of Contents**

1. Introduction
2. System Overview
3. File Structure
4. Installation
5. Usage Guide
   * Running the Main Application
   * Creating and Managing Scripts
   * Executing Test Queues
   * Viewing Logs
6. Module Breakdown
   * script\_application.py
   * main\_window.py
   * script\_generator.py
   * action\_functions.py
   * log\_handler.py
7. Troubleshooting

**Introduction**

This application is built to automate web UI interactions with wall controllers. Users input a series of commands through a graphical interface, which are then translated into predefined actions and executed in sequence. Real-time logging and queue-based task management provide a structured, user-friendly experience.

**System Overview**

* **Input Commands:** Users provide commands via the GUI.
* **Script Generation:** Commands are translated into methods.
* **Action Execution:** Methods are executed to interact with the web UI.
* **Logging:** Actions and events are logged and viewable in real-time.

**File Structure**

* script\_application.py – Main entry point.
* main\_window.py – GUI logic and user interaction.
* script\_generator.py – Converts user commands into script actions.
* action\_functions.py – Defines executable methods for UI automation.
* log\_handler.py – Manages logging and log display.

**Installation**

Follow the same steps as outlined in the Log Viewer Manual.

**Usage Guide**

**Running the Main Application**

Start the application with:

python script\_application.py

**Creating and Managing Scripts**

1. Launch the GUI.
2. Enter automation commands in the provided input field.
3. Commands are translated into actionable scripts using script\_generator.py.

**Executing Test Queues**

* Commands are queued in the order entered.
* The queue system processes each task sequentially.
* Visual feedback is provided via the GUI.

**Viewing Logs**

* Logs are displayed for each wall controller.
* Accessible via dynamically generated log tabs.

**Module Breakdown**

**script\_application.py**

* **Role:** Main execution script.
* **Function:** Initializes the GUI and links modules.

**main\_window.py**

* **Role:** Manages the PyQt5 GUI.
* **Key Features:**
  + Handles user input.
  + Updates the test queue.
  + Connects with the log handler for real-time feedback.

**script\_generator.py**

* **Role:** Translates user commands to functions.
* **Responsibility:** Interfaces with action\_functions.py to perform actions.

**action\_functions.py**

* **Role:** Houses defined methods that mimic web UI actions.
* **Examples:** Login, navigate, click buttons, extract data.

**log\_handler.py**

* **Role:** Collects and displays logs.
* **Design:** Independent logging class used by main\_window.py.

**Troubleshooting**

* **Queue Not Processing:** Ensure commands are valid and correctly formatted.
* **Web UI Not Responding:** Check network access to wall controllers.
* **Missing Logs:** Verify log\_handler is correctly initialized.

This application automates a web UI for wall controllers by processing user-defined commands and translating them into a series of scripted interactions with the UI and devices. Below is a breakdown of how the individual modules interact and contribute to the overall workflow.

**script\_application.py (Main Entry Point)**

* This is the **main script** to launch the application.
* It initializes the GUI by running main\_window.py.

**main\_window.py (Graphical User Interface)**

* Implements the **PyQt5-based GUI**.
* Allows users to:
  + Enter automation commands.
  + Queue test scripts.
  + View logs from connected wall controllers.
  + See real-time serial output (apiOutput) and test progress.
* Internally handles:
  + Starting test threads (single or queued tests).
  + Invoking script\_generator.py via subprocess.
  + Parsing and displaying serial command output in real time.

**script\_generator.py (Script Handler and Translator)**

* Responsible for **parsing the commands entered by the user**.
* Each command is matched with a predefined function via a command map.
* These functions execute interactions with the web UI or send serial commands.
* Commands like login, click, check, send\_serial are supported.

**action\_functions.py (Command Implementations)**

* Defines the **actual actions** that are performed when commands are executed.
* Example functions include:
  + Logging into the UI
  + Clicking UI elements
  + Sending serial commands to wall controllers (send\_serial\_command)
* send\_serial\_command uses asyncio to send TCP commands and return output.

**log\_handler.py (Log Viewer Support)**

* Manages the **log display system**.
* Supports real-time streaming of logs from connected devices.
* Each log tab corresponds to a device, allowing users to monitor multiple devices simultaneously.

1. **User starts the application** by running:
2. python script\_application.py
3. The **GUI launches** via main\_window.py.
4. The user **enters commands** (e.g., login admin 1234, send\_serial PWR!L001) in the command box.
5. When a test is run:
   * The GUI saves the commands to an input file.
   * script\_generator.py is called as a subprocess.
   * It processes the commands and calls appropriate functions from action\_functions.py.
6. If a command like send\_serial is executed:
   * send\_serial\_command in action\_functions.py sends the command over TCP and waits for the response.
   * The response is printed with a [SERIAL\_RESPONSE] tag.
7. The GUI listens for [SERIAL\_RESPONSE] in real time and **displays it in the apiOutput text browser**.
8. Logs from device communication are handled and displayed via log\_handler.py.

* User queues three commands:
* login admin pass123
* click settings
* send\_serial PWR!D001
* These are translated and executed:
  + login: Simulates a login via the web UI.
  + click: Navigates to the settings page.
  + send\_serial: Sends a serial command to a device.
* Output appears live in the GUI, showing API responses and test progress.

**Module Interconnection Breakdown**

This section explains how the core files of the automation application **interact with each other** and how data flows between them to control the wall controller UI.

**1. script\_application.py → main\_window.py**

* script\_application.py serves as the **entry point** for the entire application.
* It **creates and shows the main GUI window** by instantiating the MainWindow class from main\_window.py.
* Example:
* from main\_window import MainWindow

**Key Role**: Just a launcher. It hands off control to the GUI implemented in main\_window.py.

**2. main\_window.py → script\_generator.py**

* When a user **clicks to run a test**, the GUI collects:
  + The IP address
  + The list of commands
* It **writes the commands to input.txt**, then launches script\_generator.py as a **subprocess**:
* subprocess.run(["python", "script\_generator.py"], ...)
* This isolates the command processing logic and allows running scripts without blocking the GUI.

**Why this is smart**: Keeping script\_generator.py as a separate subprocess ensures the GUI stays responsive and command execution is modular.

**3. script\_generator.py → action\_functions.py**

* script\_generator.py is responsible for parsing and processing each command.
* Each command string is mapped to an actual function via a **command map dictionary**:
* command\_map = {
* "login": login\_function,
* "click": click\_function,
* "send\_serial": send\_serial\_command,
* ...
* }
* When send\_serial or another command is recognized, it directly **calls the corresponding function** imported from action\_functions.py:
* from action\_functions import send\_serial\_command, login, click, ...

**Important Flow**: The real automation logic (e.g., using Selenium or sending TCP packets) is encapsulated in action\_functions.py. This keeps logic separate from command parsing.

**4. action\_functions.py → Serial or UI System**

* This module performs the actual **low-level operations**:
  + Uses Selenium for web UI actions
  + Uses asyncio and sockets for serial/TCP communication
* For send\_serial\_command(), it:
  + Opens a connection to the device using IP and port
  + Sends a command (e.g., PWR!D001)
  + Waits for a response (e.g., ends with --END--)
  + Returns the response or an error

**Real-time feedback** is achieved by tagging output with [SERIAL\_RESPONSE], which main\_window.py filters and sends to apiOutput.

**5. main\_window.py → log\_handler.py**

* When users want to **monitor logs from multiple wall controllers**, main\_window.py uses classes and methods from log\_handler.py to:
  + Start log listener threads
  + Display logs in GUI tabs
* These logs update **in real time**, similar to how serial output updates in apiOutput.

This module helps separate logging and streaming functionality from the main GUI code, improving clarity and modularity.

| **File** | **Calls / Uses** | **Purpose** |
| --- | --- | --- |
| script\_application.py | main\_window.py | Launches GUI |
| main\_window.py | script\_generator.py (subprocess) log\_handler.py | Runs tests, manages logs |
| script\_generator.py | action\_functions.py | Parses commands and delegates execution |
| action\_functions.py | External systems (Web UI / Serial) | Executes actual actions |
| log\_handler.py | Called by main\_window.py | Displays real-time device logs |

1. **User starts the app** → script\_application.py runs → opens GUI (main\_window.py)
2. **User enters commands** → clicks run → GUI writes commands to file
3. **GUI runs script\_generator.py as a subprocess**
4. **script\_generator.py reads the commands**, processes them, and delegates each to action\_functions.py
5. If serial output is returned, it is tagged and picked up by the GUI and displayed in real time
6. **Logs** are streamed via log\_handler.py if active