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. Next Steps

**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**

QT Designer Installation:

The website used to find an installation was:

https://build-system.fman.io/qt-designer-download

move the uic.exe file into its own bin folder (for view code)

<

*python3 -m venv venv*

*call venv\Scripts\activate.bat*

*python3 -m pip install PyQt6*

>

To convert the file from .ui into python, the following command must be performed while in the venv state

< *pyuic6 -o untitled.py untitled.ui* >

Executable creation:

< *pyinstaller --onefile --windowed script\_application.py* >

**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 PyQt6 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?L) 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?D
* 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

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.

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, ...

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?L)
  + 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

Real-time Log Streaming (Threaded UI Update)

Methods involved:

* start\_log\_thread
* toggle\_log\_connection
* add\_log\_tab

How it works:

* When a user connects to a device (toggle\_log\_connection), you:
  1. Create a LogHandler for the host.
  2. If the connection is successful, you call start\_log\_thread, which starts a background thread using Thread(..., daemon=True).

Inside start\_log\_thread:

* The thread loops while the log handler is marked as connected.
* It checks the handler’s log\_queue, and if a new log is available, it appends it to the corresponding QTextBrowser.
* It sleeps briefly (time.sleep(0.1)) between checks to reduce CPU usage.

Keeps the GUI responsive while continuously displaying log updates from devices.

Methods involved:

* run\_script\_generator
* run\_single\_test\_thread
* run\_queue\_tests\_thread

How it works:

* When tests are triggered (run\_script\_generator):
  + If it's a single test, you launch run\_single\_test\_thread.
  + If there's a queue, run\_queue\_tests\_thread is used.

Inside each method:

* You start a background thread that:
  + Calls run\_single\_test(...), which performs the actual test execution.
  + Uses QtCore.QMetaObject.invokeMethod(...) to safely call UI update methods (like test\_failed, test\_progress) from the background thread.

Allows potentially long-running test scripts to run in the background, while the main GUI remains responsive and updates appropriately.

| Task | Threaded? | Main Method | Purpose |
| --- | --- | --- | --- |
| Log streaming |  | start\_log\_thread | Stream logs live into QTextBrowser without freezing UI |
| Single test execution |  | run\_single\_test\_thread | Run a script in background, show results in GUI |
| Queue test execution |  | run\_queue\_tests\_thread | Run a series of scripts sequentially in a thread |

You're manually managing threads with Python's threading.Thread, which works, but be mindful that:

* GUI updates must always happen on the main thread. You're mostly safe using invokeMethod and QTextBrowser.append, but be cautious.
* Consider using QThread and signals/slots for cleaner integration with PyQt's event loop, especially if complexity grows.

Purpose of LogHandler

This class connects to a remote device over Telnet, sends a command to start tailing a log file, and reads incoming log data asynchronously using asyncio. The log data is passed to your app using a thread-safe queue (Queue), so your PyQt GUI can read it safely from another thread.

Class Components

\_\_init\_\_

def \_\_init\_\_(self, host, port=23):

Initializes the handler:

* host, port: Target device and port for Telnet.
* reader, writer: Telnet stream objects.
* stop\_event: Used to signal the reading thread to stop.
* log\_queue: Stores incoming log lines for the GUI to consume.
* connected: Whether the connection is active.
* read\_thread: The thread running the async event loop.
* loop: The asyncio event loop for reading.
* \_running: Flag to keep the reading loop going.

Connection Logic

async\_connect

async def async\_connect(self):

* Tries to open a Telnet connection using telnetlib3.open\_connection.
* On success, sets connected = True.
* Sends the command tail -f /var/log/messages to stream logs.
* Runs asynchronously (non-blocking).

Reading Log Data

read\_loop

async def read\_loop(self):

* Continuously reads incoming data while \_running and stop\_event aren't triggered.
* Each chunk of data is put into the log\_queue.
* This loop runs in the background thread's asyncio event loop.

Thread Entry Point

run\_async\_loop

def run\_async\_loop(self):

* This function is run inside a background thread.
* It:
  1. Creates a fresh asyncio event loop.
  2. Connects to the device.
  3. Starts the read\_loop.
  4. Cleans up resources when done.

Connecting and Starting the Thread

connect

def connect(self):

* Starts the reading thread (run\_async\_loop runs inside it).
* Waits briefly for the connection to establish (self.connected becomes True).
* Returns True if connection is successful.

Disconnecting

disconnect

def disconnect(self):

* Stops the reading loop by setting \_running = False and stop\_event.
* Joins the thread to clean up.
* Resets all state variables to prepare for reconnect.

Sending Commands

send\_command

def send\_command(self, command):

* Sends a command over the Telnet connection.
* Uses asyncio.run\_coroutine\_threadsafe to run an async send inside the thread's event loop.
* This ensures thread safety and doesn't block the main thread.

How It All Works Together

1. User clicks "Connect" in the GUI.
2. toggle\_log\_connection calls LogHandler.connect().
3. connect():
   * Starts a background thread that runs run\_async\_loop().
   * That launches an asyncio event loop in the thread.
   * It connects and begins streaming logs via read\_loop().
4. Each log message is pushed into log\_queue.
5. A separate UI thread periodically reads from log\_queue and updates the GUI.

Threading + Asyncio Summary

Combining threading and asyncio like this:

| Thread | Role |
| --- | --- |
| Main Thread | PyQt GUI |
| Log Thread | Runs run\_async\_loop() → manages asyncio loop and Telnet read |
| Asyncio Tasks | Handle Telnet connection and read\_loop() |

| **Command** | **Function Called** | **Description** |
| --- | --- | --- |
| login | login | Logs the user into the web UI by entering credentials and navigating to the dashboard. |
| logout | logout | Logs the user out of the web UI and returns to the login screen. |
| power\_cycle | power\_cycle | Toggles the power state of the device off and then back on (or vice versa). Used for hardware reset behavior. |
| check | check\_element | Verifies if a specific element is present and correctly configured on the current page. |
| screenshot | screenshot | Takes a screenshot of the current page for documentation or comparison purposes. |
| check\_button | check\_button | Checks the status (enabled, active, etc.) of a specific button on the page. |
| click | click | Simulates a click on a specified element or button within the web UI. |
| power | power\_toggle | Directly toggles the device’s power (without a full cycle), such as turning it on or off. |
| check\_text | check\_text | Verifies if a particular element contains the expected text. |
| wait | wait | Pauses script execution for a specified duration (e.g., "wait 5 seconds"). Useful for timing and syncing. |
| highlight | highlight | Temporarily highlights an element visually to indicate focus or for debugging. |
| tile\_pattern | set\_tile\_pattern | Applies a specified tile pattern configuration to the wall display (layout or image tiling). |
| image\_compare | compare\_images | Compares a current screenshot with a reference image to validate UI correctness visually. |
| element\_compare | compare\_elements | Compares the visual or structural representation of one UI element with a saved reference. |
| send\_serial | send\_serial\_command | Sends a direct serial (TCP/IP) command to the device and processes the response, such as power status or diagnostic commands. |

**What is configuration.hjson?**

The configuration.hjson file acts as the centralized page and UI element map for your automation framework. It's written in HJSON — a more human-readable form of JSON — and it typically defines:

* The pages in the web UI (e.g. "dashboard", "settings", "power\_page")
* The elements present on each page (buttons, text fields, labels, etc.)
* Their selectors (XPath, CSS, IDs, etc.)
* Any metadata (labels, expected values, input types)

This config allows your automation to adapt dynamically to the structure of the web UI without hardcoding UI paths in your scripts.

**How the Files Use configuration.hjson**

| **File** | **How It Uses configuration.hjson** |
| --- | --- |
| action\_functions.py | Most of the action functions (like click, check, highlight, check\_button, etc.) read from configuration.hjson to locate and interact with UI elements. |
| script\_generator.py | Translates a high-level command (like check power\_status) into a call to check\_element(driver, "power\_status", page, config), where config comes from configuration.hjson. |
| main\_window.py | Indirectly relies on the configuration when invoking script execution and displaying results based on whether actions succeed or fail. |

**Adding a New UI Element**

Suppose you're adding a new button called restart\_button to the "power\_page" in the web UI. Here's how you’d update your application:

**Step 1: Update configuration.hjson**

Assume your HJSON looks like this:

power\_page: {

power\_toggle: {

selector: "#powerToggle"

type: "button"

label: "Power"

}

}

You would add your new element:

power\_page: {

power\_toggle: {

selector: "#powerToggle"

type: "button"

label: "Power"

}

restart\_button: {

selector: "#restartButton"

type: "button"

label: "Restart"

}

}

**Step 2: Use the Element in a Command**

In your test script or command queue, add:

click restart\_button

**Step 3: Let the Automation Handle It**

Here’s what happens:

1. script\_generator.py reads click restart\_button → identifies click as the action and restart\_button as the target.
2. It uses the current page context (e.g., "power\_page") and looks up restart\_button in configuration.hjson.
3. click() from action\_functions.py receives the driver, the element name (restart\_button), the page, and the full config.
4. It pulls the selector from the config, finds the element on the page using Selenium, and clicks it.

All of this happens dynamically — no need to modify any Python code if the config is correct.

* You can add, remove, or edit UI elements in configuration.hjson without changing your core Python code.
* This makes the system extensible, maintainable, and scalable as the web UI evolves.
* Your command scripts stay high-level and human-friendly (e.g., click submit\_button), and the configuration handles the technical details.

**Future Steps**

The elementNamesButton QT button should be used to display all of the contents from the configuration.hsjon file so that the user can view the names directly. Also, a visual tool can be used to display each page and label what each element’s name corresponds to.