ChatGPT

replit_agent_prompt.txt

Industrial Automation Blog-Lab Site (Flask)

Project Overview

This project is to build a dynamic website for industrial automation engineers using the Flask web framework. The site will serve as a **blog and lab hub** featuring technical posts, hands-on protocol testing labs, tool documentation, and how-to guides (covering Python scripting, Docker usage, Wireshark analysis, etc.). Key goals include a modular structure, easy content management (using Markdown for posts/labs), and future extensibility (forum integration and vendor-specific content).

Key features to implement:

- A **blog section** with posts on industrial automation topics (e.g. Docker, Wireshark, OT/IT basics).
- An interactive **labs section** with repeatable protocol labs (Modbus, CIP, OPC UA, MQTT, Siemens S7, DNP3, IEC 60870-5-104), each including DI/DO/AI/AO, diagnostics, and timer scenarios.
- A **tools & libraries section** providing documentation and guidance on using relevant Python libraries (for Modbus, CIP, OPC UA, etc.) and tools.
- Content written in **Markdown** for easy editing, rendered to HTML in the Flask app.
- A **modular project structure** (`templates/`, `static/`, `content/` or `labs/`, etc.) to organize code and content.
- A stub for a **forum section** (hidden from nav until ready) to allow future community discussions.
- **"Coming soon" placeholders** for vendor-specific sections (e.g. Honeywell Experion DCS, Yokogawa CENTUM DCS), indicating planned future content.
- Clear navigation and directory organization for maintainability.

Technical Requirements

1. **Framework & Language:** Use **Python 3.x** and **Flask** (a beginner-friendly web framework) to build the web application. Ensure the app can run easily (e.g. via `flask run` or a simple `python app.py` entry point).

2. **Content Management (Markdown):** Store site content (blog posts, lab instructions, tool docs) as Markdown files on disk. The Flask app will load these files and render them as HTML for display [1]. This allows non-developers to write or edit content easily without touching HTML. You may use a library like **Markdown** (Python-Markdown) or an extension like **Flask-FlatPages** to facilitate Markdown rendering to HTML [2].

3. **Project Structure:** Organize the project in a **modular directory structure**. For example:

```
project_root/
├── app.py                # Flask app initialization and route
definitions
├── requirements.txt      # project dependencies (Flask, Markdown, etc.)
```

```
├── content/              # directory for all Markdown content files
│   ├── blog/             # markdown files for blog posts
│   ├── labs/             # markdown files for lab guides (organized by
protocol)
│   └── tools/            # markdown files for tools & libraries
documentation
├── templates/            # Jinja2 HTML templates
│   ├── base.html         # base layout template (nav, footer, etc.)
│   ├── index.html        # home page template
│   ├── blog/             # blog-related templates
│   │   ├── list.html     # template to list blog posts
│   │   └── post.html     # template for an individual blog post
│   ├── labs/             # labs-related templates
│   │   ├── list.html     # template to list protocols/labs
│   │   └── lab.html      # template for individual lab content
│   ├── tools/            # tools & libraries templates
│   │   └── tool.html     # template for tool/library detail page
│   └── forum/            # forum templates (stub)
│       └── forum.html    # placeholder page for forum
└── static/               # static files (CSS, JS, images)
    ├── css/             # stylesheets
    ├── js/              # scripts
    └── images/          # images for use in posts or layout
```

This structure separates content from code and HTML. Flask will serve static files from `static/` and use templates from `templates/`. Markdown files in `content/` (or a similarly named folder) will be read and converted for display.

4. **Routing and Blueprints:** Implement **routes** for each section:

5. **Home page:** (`/`) A simple welcome page (in `index.html`) that links to the main sections (Blog, Labs, Tools). It can briefly describe the site's purpose.
6. **Blog:**
   - List view (`/blog`) that reads all Markdown files in `content/blog/` and displays a list of posts (title, date, snippet).
   - Post view (`/blog/<slug>`) that loads a specific Markdown file and renders it via `post.html`.
7. **Labs:**
   - Overview (`/labs`) listing each protocol available (Modbus, CIP, OPC UA, etc.). This page should link to each protocol's lab index or first lab.
   - Protocol lab index (`/labs/<protocol>`) that shows that protocol's available labs (e.g. "Modbus Labs: Lab 1 – DI/DO, Lab 2 – AI/AO, …"). Alternatively, if only a few labs, you might link directly from `/labs` to each lab page.
   - Individual lab pages (`/labs/<protocol>/<lab_name>`) that load the corresponding Markdown from `content/labs/<protocol>/<lab_name>.md` and render it.
8. **Tools & Libraries:**
   - List page (`/tools`) summarizing all tools/libraries with brief descriptions.
   - Detail pages (`/tools/<tool_name>`) rendering the Markdown content for that tool or library.

9. **Forum:**
    - Stub page (`/forum`) that is currently not active. This can return a template that says "Forum coming soon" and might be excluded from nav.

10. Consider using **Flask Blueprints** to organize routes for each section (e.g. a blueprint for blog, one for labs, etc.), which helps keep the `app.py` uncluttered. This is optional for a smaller app, but beneficial as content grows.

11. **Markdown Rendering:** Use a Markdown library to convert content to HTML. For example, you can install `markdown` (the Python Markdown package) and do something like:

    ```
    import markdown
    html = markdown.markdown(markdown_text, extensions=['fenced_code',
    'tables'])
    ```

    Integrate this into your route handlers or use Flask-FlatPages which can simplify serving Markdown content. The content files can include YAML front-matter for metadata (like title, date) if needed (FlatPages supports this). Ensure that any HTML generated is safe (consider using Markdown's safe_mode or Bleach for sanitizing if you allow user-contributed content in future).

12. **Base Template & Navigation:** Create a `base.html` template that includes a top navigation bar with links to **Blog**, **Labs**, **Tools & Libraries**, and placeholders for **Forum**, etc. Also include links or icons for **GitHub** and **DockerHub** (to integrate those resources). Each page template will extend `base.html`. The nav should highlight the current section and clearly indicate any "Coming Soon" sections (e.g., grey out or add a tooltip for forum if it's disabled).

13. **Blog Implementation:**

14. Store blog posts as Markdown files under `content/blog/` (e.g. `docker_intro.md`, `wireshark_basics.md`, etc.). Include metadata at top of each file (title, date, author if needed).
15. The `/blog` route will collect all files in `content/blog/`, sort them by date or name, and pass them to `blog/list.html` template. This template will loop through posts and display their title, date, and a short excerpt (you can define an excerpt as the first N characters or up to a separator in the Markdown).
16. The `/blog/<slug>` route will load the specific Markdown file matching the `<slug>` (file naming convention should correspond to slug). It then renders `blog/post.html` with the post content (converted to HTML) and metadata.

17. Ensure code snippets in blog Markdown render with proper formatting (you can include a CSS for code highlighting, or use Pygments via Markdown extension for syntax highlighting).

18. **Labs Implementation:**

19. **Content Structure:** Under `content/labs/`, create a subfolder for each protocol (e.g., `modbus/`, `cip/`, `opcua/`, `mqtt/`, `s7/`, `dnp3/`, `iec104/`). Within each, create Markdown files for each lab scenario: for example, `di_do.md` (digital I/O lab), `ai_ao.md` (analog I/O lab), `diagnostics.md`, `timer.md` for timing or clock sync. This yields a uniform structure for labs across protocols.

20. **Lab Pages:** Each lab Markdown should be written as a step-by-step tutorial. For instance, **Modbus DI/DO lab** might explain how to read a digital input (coil) and write to a digital output (coil) using Python (Pymodbus), with code samples. **Diagnostics** might cover reading device identity or exception codes. **Timer** could involve using a real-time clock register or simulating scheduling.
21. The `/labs` route shows an index of protocols. For each protocol, list the labs available. Example:
    - *Modbus Labs:* Lab 1 – Digital I/O, Lab 2 – Analog I/O, Lab 3 – Diagnostics, Lab 4 – Timer.
    - *CIP Labs:* Lab 1 – Digital I/O (reading/writing PLC tags), Lab 2 – Analog I/O, Lab 3 – Device Diagnostics, Lab 4 – Clock Sync.
    - etc.
22. The `/labs/<protocol>/<lab_name>` route loads the corresponding markdown file. Use a generic `labs/lab.html` template to display a lab. This template can include the lab title, and the content (converted from Markdown). It might also include navigational links to other labs in the same protocol or a link back to the protocol index.

23. **Consistency:** Make lab content structured similarly for each protocol (as much as makes sense) so users can carry knowledge from one to another. For example, every protocol's "Digital I/O" lab will involve a similar task (read a boolean input and write an output) but using that protocol's library. The idea is to give a comparative understanding of how each protocol handles similar tasks.

24. **Tools & Libraries Section:**

25. Under `content/tools/`, create Markdown files for each tool or library the site covers. Each file should describe *what the tool/library is*, *how to install it*, and *basic usage examples* relevant to industrial automation.
26. For example:
    - `modbus.md` might describe **Pymodbus** (the Python Modbus library) – how to use it to connect to a Modbus server, read registers, etc. Include code snippets.
    - `cip.md` for **pycomm3 or cpppo** – how to connect to an Allen-Bradley PLC using CIP/EtherNet-IP, read/write tag values, etc.
    - `opcua.md` for **python-opcua** – how to set up an OPC UA client and server, browse nodes, read/write values.
    - `mqtt.md` for **Paho MQTT** – how to publish/subscribe to topics.
    - `s7.md` for **Snap7** – how to connect to a Siemens PLC and read/write memory or DB values.
    - `dnp3.md` for **PyDNP3** – how to use DNP3 in Python.
    - `iec104.md` for **IEC 60870-5-104** – how to use a Python library to simulate or communicate via IEC104.
27. Each of these pages should reference the library's official documentation or description for accuracy. For instance, note that *Pymodbus is a full Modbus protocol implementation offering both client and server functionality in Python*, or that *pycomm3 is a Python library for communicating with Allen-Bradley PLCs over EtherNet/IP (the Common Industrial Protocol)* [3], etc. (See **Citations** section below for more info to include.)
28. Also include tools like **Docker** and **Wireshark**:
    - A page for **Docker** could explain how to use Docker images to simulate PLCs or run test environments (for example, using an Eclipse Mosquitto container for an MQTT broker, or a Modbus simulator container).
    - A page for **Wireshark** could give tips on capturing industrial protocol traffic and using Wireshark dissectors for Modbus, CIP, etc., including filters and example captures.

29. The `/tools` route will show a list of all these items (Modbus/Pymodbus, CIP/pycomm3, OPC UA/python-opcua, MQTT/Paho, S7/Snap7, DNP3/PyDNP3, IEC104, Docker, Wireshark, etc.) possibly grouped by category (protocol libraries vs general tools).
30. Each item in the list links to `/tools/<name>` which renders the respective Markdown content in `tool.html`.
31. **GitHub & DockerHub Integration:**
    - Provide prominent links to **GitHub** and **DockerHub**:
    - The **GitHub repository** likely contains the site's source code and possibly code examples for labs. In the nav bar, include a GitHub icon or link labeled "GitHub" that opens the project repo.
    - The **DockerHub** link could point to a user or org page hosting Docker images for the labs. For instance, if there are Docker images for a Modbus simulator or an OPC UA server for testing, host them on DockerHub and link to them. Label this link "DockerHub".
    - These are external links; mark them to open in a new tab (use `target="_blank"` in anchor or a note that they leave the site).
32. **Forum Stub:** Though a forum is planned for the future, set up a basic stub:
    - In `templates/forum/forum.html`, put a placeholder message (e.g., **"Community Forum – Coming Soon"**).
    - Create a route `/forum` that simply renders this template. Do not list "Forum" in the main navigation (or if you do, disable the link or add a coming-soon note) until the forum is implemented.
    - Document in code comments that a forum system could be integrated here (for example, using Flask extensions or an embedded external forum like Discourse or a simple Flask chat app).
    - This prepares the ground for adding forum functionality later without restructuring.
33. **Coming Soon Placeholders (Vendor-Specific):**
    - Add pages or sections to acknowledge upcoming vendor-specific content:
    - **Honeywell Experion** and **Yokogawa CENTUM** are two distributed control systems (DCS) to earmark. For each, create a Markdown page (e.g., `content/labs/honeywell.md` and `content/labs/yokogawa.md`) or simply include them in the labs index as placeholders.
    - These pages can contain a brief message such as "*Honeywell Experion (Honeywell's DCS platform) content is coming soon.*" and possibly outline what will be included (like connection guides, specific tools or libraries for that system, etc.).
    - In the **Labs** section or a separate **Vendors** section, list these with a "Coming Soon" label. For example, on the labs index page, after the protocol labs, you might list "Honeywell Experion – Coming Soon" and "Yokogawa CENTUM – Coming Soon".
    - Make it clear these are stubs. You can still create routes for them (`/labs/honeywell` or `/vendors/honeywell`) to show a placeholder page if visited.
    - By structuring it now, you leave a slot to later fill in content for vendor-specific tutorials (which might involve proprietary systems or specific software like Honeywell's Experion PKS or Yokogawa's CENTUM DCS).

34. **Extensibility for Future Sections:**

    - Keep the door open for new sections such as:
    - **Ansible Automation Examples:** e.g., using Ansible to deploy or configure industrial devices, automate network setups, etc.
    - **ICS Security Testing:** e.g., labs or guides on security tools (like using Wireshark for intrusion detection, Kali Linux tools for penetration testing in OT environments, etc.).
    - These are not implemented now, but note in documentation or comments that the site structure (especially the content directory and nav) can be expanded. For instance, an "Ansible" section could be another top-level menu or a subsection under Labs or Tools.
    - You might include these in a roadmap on the home page or a dedicated "Coming Soon" page.

35. **Directory Structure and File Organization:** (Recap) The project should be organized for clarity:

    - **Flask App Entry**: `app.py` sets up Flask, config (for FlatPages or Markdown), and registers blueprints or routes.
    - **Templates**: organized in subfolders for each section (as outlined above).
    - **Static files**: CSS/JS for styling. Consider using a simple CSS framework (Bootstrap) to save time on design.
    - **Content**: Markdown files in a structured content directory. Keeping these outside the templates means content can be written and managed separately from code.
    - Ensure this structure is documented (you can add a section in the README or comments explaining what goes where, so a new contributor or the Replit agent itself can navigate it).

36. **Dependencies:** In `requirements.txt`, list needed packages:

    - `Flask` (the appropriate version).
    - Markdown rendering library: either `Markdown` (Python-Markdown) or `Flask-FlatPages` (which itself uses Markdown) [2], plus `Frozen-Flask` if you ever plan to generate a static version.
    - Libraries for code examples: e.g., `pymodbus`, `pycomm3`, `opcua`, `paho-mqtt`, `snap7`, `pydnp3`, `iec104` etc., if you want to have them available for testing code in labs. (These might not be strictly required to run the site, but including them could be useful if, say, a lab page runs a live demo or test. It's optional to include them all; you can just instruct the user to install those as needed.)
    - Any other utilities (maybe a syntax highlighting extension for Markdown).

37. **Citations & Accuracy:** When describing tools and libraries on the site, ensure the information is accurate. Cite version numbers or official descriptions where relevant. For example:

    - *Pymodbus* – note that it supports Modbus TCP/RTU and has client/server implementations.
    - *Pycomm3* – mention that it started as a fork of `pycomm` for AB PLC communication and supports reading/writing tags and PLC time services [4].
    - *Python OPC UA (FreeOPCUA)* – describe it as a pure Python implementation of OPC UA (client & server) [5].
    - *Eclipse Paho MQTT* – mention it's the standard Python MQTT client library supporting MQTT 3.1.1 and 5.0 [6].
    - *Snap7* – note it's a wrapper around Snap7 library for Siemens S7 communication [7].
    - *PyDNP3* – explain it's a Python binding to the open source DNP3 stack (OpenDNP3) [8].

- *iec104-python* – mention it provides a high-level interface to simulate SCADA/RTU communication for IEC 60870-5-104 [9].
- Including such details lends credibility and helps users find those libraries.

38. **Testing:** After building, thoroughly test the site:
    - Verify each route returns the expected content.
    - Test Markdown rendering for various elements (headings, lists, code blocks, tables) by creating sample content.
    - If possible, test some lab instructions by running the code snippets (for example, ensure a Modbus lab's code works with a Modbus simulator).
    - Check that navigation links work and that "coming soon" sections are clearly marked to avoid confusion.
39. **Use Case Validation:** Keep the target audience in mind – industrial automation engineers who may not be web experts. The site should be straightforward to navigate:
    - The **Blog** provides conceptual knowledge and general tutorials.
    - The **Labs** provide practical, hands-on experience with each protocol, in a stepwise manner.
    - The **Tools & Libraries** section serves as a reference or cheat-sheet for how to use specific technologies (like a quick start guide for each library/tool).
    - Ensure the tone of writing in content is approachable and not overly assumed in prior knowledge (explain acronyms like OT vs IT, etc., where necessary).
40. **Replit-specific setup:** If using Replit to host/run this:
    - Add a `replit.toml` if needed, specifying the run command (`flask run` or `python app.py`).
    - Ensure the Flask app is configured to listen on the correct host/port for Replit (Replit typically uses `$PORT` env var).
    - Possibly prepopulate some content so the site isn't empty on first run (e.g., one sample blog post, one lab, etc. as examples).
41. **Documentation & Comments:** Provide clear comments in code and maybe a `README.md` for the project:
    - Explain how content can be added (e.g., "To add a new blog post, create a Markdown file in content/blog and it will automatically appear on the Blog page").
    - Explain how to extend (e.g., "To add a new protocol lab, create a folder in content/labs and add your .md files; update labs index if needed").
    - These instructions will make it easier for others to contribute or for the project to evolve.

By following these guidelines, the Replit agent (or a developer) should be able to generate a Flask-based **"blog-lab" web application** that is well-structured, easy to update (thanks to Markdown content), and rich in valuable content for industrial automation professionals.

Citations for Library References (for inclusion in site content)

- **Pymodbus (Modbus):** "Pymodbus is a full Modbus protocol implementation offering a client and server with synchronous/asynchronous API and simulators."

- **Pycomm3 (CIP/EtherNetIP):** Pycomm3 is a Python 3 library for communicating with Allen-Bradley PLCs using Ethernet/IP (CIP) [3], supporting services like reading/writing tags and getting/setting the PLC time [4].
- **Python OPC UA (FreeOPCUA):** The `python-opcua` library provides the necessary tools and functionalities for interacting with OPC UA servers and clients in Python [5].
- **Eclipse Paho MQTT:** The Paho Python Client provides a class with support for MQTT v5.0, v3.1.1, and v3.1, making it straightforward to publish/subscribe messages from Python [6].
- **Snap7 (Siemens S7):** *python-snap7* is a Python wrapper for the open-source Snap7 library, a multi-platform Ethernet communication suite for interfacing natively with Siemens S7 PLCs [7].
- **PyDNP3 (DNP3):** Python bindings for the OpenDNP3 library (an open-source implementation of DNP3), enabling Python-based DNP3 protocol communication [8].
- **iec104-python:** Provides an object-oriented Python module to simulate SCADA systems and RTUs communicating via the IEC 60870-5-104 protocol [9].

(These citations should be used on the site's Tools & Libraries pages to give credit and allow users to find more information.)

---

site_sitemap.txt

Home – *Welcome page introducing the blog-lab site and its purpose, with links to major sections.*

Blog – *Technical articles and guides on industrial automation topics:* - **Docker for Industrial Automation Labs** – (Guide on using Docker to set up OT simulation environments, containerizing PLC simulators, etc.) - **Wireshark for ICS Traffic Analysis** – (Tutorial on capturing and analyzing Modbus, CIP, etc. packets with Wireshark.) - **OT/IT Networking Basics** – (Explainer on Operational Technology vs Information Technology networking concepts in industry.) - **Python Scripting for PLCs** – (Tips on using Python to communicate with PLCs and other devices, bridging the OT-IT gap.) - **MQTT Basics in IIoT** – (Introduction to MQTT protocol and how it's used for IIoT data publishing/subscribing.)

(Additional blog posts on relevant topics can be listed here as the content grows.)

Labs – *Hands-on protocol testing labs (each protocol has multiple scenarios):* - **Modbus Protocol Labs:** - *Lab 1: Digital Input/Output Control* – Using Modbus (Pymodbus) to read coil status and write to coils (turn outputs on/off). - *Lab 2: Analog Input/Output Operations* – Reading analog input registers and writing analog output registers (e.g., sensor values, setpoints). - *Lab 3: Diagnostics & Error Handling* – Using Modbus diagnostics function and interpreting exception codes. - *Lab 4: Timing and Polling* – Implementing a simple polling loop and simulating a real-time clock read via Modbus. - **CIP (EtherNet/IP) Labs:** - *Lab 1: Digital I/O with CIP* – Reading and writing PLC tags (bits/booleans) on an Allen-Bradley Logix controller via pycomm3 [4]. - *Lab 2: Analog Data via CIP* – Reading analog values (e.g., process variables) and writing outputs using CIP messages. - *Lab 3: Device Identity & Diagnostics* – Using CIP services to get device information and status (identity object, error codes). - *Lab 4: Clock Synchronization* – Getting and setting the PLC's clock over CIP (time services) [4]. - **OPC UA Labs:** - *Lab 1: OPC UA Client-Server Basics* – Setting up a Python OPC UA server and client, browsing the address space, reading/writing node values. - *Lab 2: Subscription Handling* – Subscribing to data change events and monitoring values updates from an OPC UA server. - *Lab 3: Security Configuration* – Connecting with security (certificates, encryption) in OPC UA, exploring user authentication. - *Lab 4: Method Invocation* – Calling methods on an OPC UA server from a client (e.g., remote procedure calls in an OPC UA model). - **MQTT Labs:** - *Lab 1: Publish/Subscribe Basics* – Using Eclipse Paho to connect to an MQTT broker, publish sensor readings, and subscribe to a topic for updates. - *Lab 2: MQTT Quality of Service* – Experimenting with QoS levels 0, 1, 2 and retained messages to see their effects. - *Lab 3: Secure MQTT Communication* – Setting up

an MQTT broker with TLS and authenticating a client with username/password or certificates. - *Lab 4: Bridging OT Data via MQTT* – Simulating a scenario where a PLC's data is published to IT systems using MQTT (bridging an OT device to a cloud service). - **Siemens S7 Protocol Labs:** - *Lab 1: Connecting to S7 PLC* – Using python-snap7 to connect to a simulated S7 PLC and read a data block (memory area) [7]. - *Lab 2: Digital/Analog I/O with S7* – Reading digital inputs/outputs and analog values (e.g., using Snap7 to get PLC I/O states or flags). - *Lab 3: PLC Diagnostics* – Reading diagnostic data or error stack from a Siemens PLC via S7 protocol. - *Lab 4: PLC Time Sync* – Reading the PLC's system time and writing a new time to the PLC (if supported) to understand time synchronization. - **DNP3 Labs:** - *Lab 1: DNP3 Outstation Simulation* – Using PyDNP3 to set up a simple outstation and master, and read binary/analog points (telemetry) [8]. - *Lab 2: Control Operations* – Sending control commands (operate relay, etc.) from a DNP3 master to an outstation and handling responses. - *Lab 3: Unsolicited Reporting* – Configuring an outstation to send unsolicited responses and having the master process them. - *Lab 4: Secure DNP3 (SA)* – Overview of DNP3 Secure Authentication features (discussion or minor demo if libraries support it). - **IEC 60870-5-104 Labs:** - *Lab 1: IEC104 Client/Server Basics* – Establishing a connection between a client and server (RTU simulator) and reading single-point information [9]. - *Lab 2: General Interrogation* – Executing a general interrogation command to retrieve all station data from an RTU (common IEC104 procedure). - *Lab 3: Command Transmission* – Sending a command (operate) from the control center to the RTU and observing the process (with acknowledgments). - *Lab 4: Time Synchronization* – Using IEC104 time sync mechanism to synchronize the RTU's clock with the control center. - **Honeywell Experion DCS – Coming Soon** – *(Placeholder for future labs or guides on Honeywell's Experion PKS DCS platform.)* - **Yokogawa CENTUM DCS – Coming Soon** – *(Placeholder for future labs or guides on Yokogawa's CENTUM VP DCS platform.)*

(Lab sections marked "Coming Soon" indicate planned content for vendor-specific systems that will be added in the future.)

Tools & Libraries – *Reference pages for each protocol library and tool:* - **Modbus – Pymodbus:** Pymodbus is a Python library that implements the Modbus protocol (client and server) for TCP, RTU, etc., enabling reading and writing of PLC registers/coils. *(Includes install instructions and example code for connecting to a Modbus device.)* - **CIP (EtherNet/IP) – pycomm3:** pycomm3 is a Python library for Allen-Bradley PLC communication over EtherNet/IP (CIP), allowing read/write of controller tags and other services [3]. *(Includes example of reading a tag and writing a value on a Logix PLC.)* - **OPC UA – python-opcua (FreeOPCUA):** A pure Python OPC UA client/server library. It provides tools to create an OPC UA server or connect as a client to read/write data and call methods on OPC UA nodes [5]. *(Page includes a basic example of setting up a server and reading a value.)* - **MQTT – Eclipse Paho MQTT:** The standard Python MQTT client library that supports MQTT v3.1.1 and v5.0. It allows publishing messages to topics and subscribing to topics easily in Python [6]. *(Includes example code to connect to a broker and send/receive a message.)* - **Siemens S7 – Snap7 (python-snap7):** Python-snap7 is a wrapper for the Snap7 library, which enables native Ethernet communication with Siemens S7 PLCs (S7-300/400/1200/1500) [7]. *(Includes example to read a DB (data block) value from a PLC.)* - **DNP3 – PyDNP3:** PyDNP3 provides Python bindings to the OpenDNP3 C++ library, allowing you to implement DNP3 masters or outstations in Python [8]. *(Includes example snippet to initialize a DNP3 master, add an outstation, and read a point.)* - **IEC 60870-5-104 – iec104 (IEC104-python):** A Python module (often called *c104*) that can simulate SCADA/RTU communication via the IEC 60870-5-104 protocol [9]. *(Includes a brief example of starting an IEC104 server and client and sending a test command.)* - **Docker** – *Docker container usage for labs:* Guidance on using Docker to run simulation environments. *(E.g., how to start a Modbus simulator container, an MQTT broker container, and how to network containers for testing.)* - **Wireshark** – *Packet analysis tool:* Tips for capturing industrial protocol traffic and using Wireshark's dissectors. *(E.g., how to apply a filter for Modbus TCP (`tcp.port == 502`) or follow a CIP connection, etc.)* - **GitHub Repository** – *(Link)* The official GitHub repository containing this site's source code, sample code for labs, and configuration

files. Users can browse the code or contribute. - **DockerHub Profile** – *(Link)* DockerHub page where pre-built Docker images for some labs are hosted (e.g., simulation containers for various protocols).

Forum – *Community Forum (disabled for now):* A placeholder for a future discussion forum. *(The forum section is not yet active; once launched, it will allow users to discuss industrial automation topics, ask questions, and share knowledge.)*

Future Sections (Planned): - **Ansible Automation** – *Coming Soon:* Examples of using Ansible to automate industrial infrastructure (e.g., deploying configurations to devices, managing network gear in OT). - **ICS Security Testing** – *Coming Soon:* Labs and guides focused on security in industrial control systems (penetration testing exercises, defense mechanisms, using security tools in an OT context). - *(Any additional future expansions can be listed here, such as new protocols, cloud integration, machine learning for IoT, etc., as the site evolves.)*

---

1 Build a Simple, Static, Markdown Blog with Flask - James Harding
https://jameshard.ing/posts/simple-static-markdown-blog-in-flask/

2 Building A Lightweight Blogging CMS In 10 Lines of Code - DEV Community
https://dev.to/steventey/building-a-lightweight-blogging-cms-in-10-lines-of-code-e1a

3 4 GitHub - ottowayi/pycomm3: A Python Ethernet/IP library for communicating with Allen-Bradley PLCs.
https://github.com/ottowayi/pycomm3

5 Unlocking the Power of OPC UA for Industrial Automation with Python: A Comprehensive Guide Part-1 | by Fazlul Karim | Medium
https://medium.com/@kfazlul.fahim/industrial-automation-with-python-and-opc-ua-comprehensive-guide-part-1-16e2f0875d51

6 Eclipse Paho | The Eclipse Foundation
https://eclipse.dev/paho/clients/python/

7 python-snap7 · PyPI
https://pypi.org/project/python-snap7/

8 ChargePoint/pydnp3: Python bindings for opendnp3 libary - GitHub
https://github.com/ChargePoint/pydnp3

9 Welcome to iec104-python's documentation! — iec104-python 2.2 documentation
https://iec104-python.readthedocs.io/latest/