PROJECT DEVELOPMENT JOURNAL: TempServerMAKER

This log documents the transition of the TempServerMAKER application from a single, monolithic Python script to a robust, client-server application with a full test suite.

Phase 1: The Monolithic Script

• Goal: Create a simple, portable, single-file server that could generate a self-contained HTML file dump of a project directory.
• Implementation: A single app.py script contained hardcoded Python strings for all the HTML, CSS, and JavaScript required for the front-end.
• Problem: This approach, while simple, made front-end development extremely difficult. Editing CSS or JavaScript required careful escaping of characters within Python strings, leading to bugs and a cumbersome workflow. The script was hard to read and maintain.

Phase 2: Refactoring to a Client-Server Architecture

• Goal: Separate the back-end (Python server logic) from the front-end (UI and user interaction) to improve maintainability and ease of development.
• Process:
1. Created three new files in a _src directory: index.html, style.css, and index.js.
2. Modified app.py to stop generating its own HTML. Its new role was to:
a. Act as a web server for the static front-end files.
b. Read index.html as a template.
c. Gather all the project file data.
d. Dynamically inject the file data as JSON into empty <script> tags in the template before serving it to the browser.

Phase 3: Debugging the New Architecture

• Problem 1: "index.html not found" error upon starting the server.
• Cause: The script was being run from the project root, but the front-end files were in the _src subdirectory. The script's default working directory was incorrect.
• Solution: Modified app.py to be location-aware. It was updated to use Path(__file__).parent as its default serving directory, ensuring it always found its own front-end files.
• Problem 2: The web page loaded, but the UI was empty. The file tree and content cards did not appear.

• Cause: A single, extra closing brace } at the end of index.js caused a fatal syntax error, which prevented the browser from executing any of the JavaScript.
• Solution: The syntax error was identified and removed. We then restored the correct JavaScript logic to render the hierarchical file tree and the expandable content cards, matching the UI of the original monolithic version.

Phase 4: Re-integrating and Testing the API

• Goal: Add back the programmatic API hooks (/ping, /files, /shutdown, etc.) for external tools while keeping the new client-server structure.
• Process: The API handler logic from the original monolithic script was carefully merged into the new, refactored app.py.
• Testing:
1. A new test file, tests/test_smoke.py, was created to validate the API endpoints.
2. Problem: The initial tests failed with an AttributeError, indicating the server object didn't exist when the test tried to run it.
3. Cause: The test was attempting to start the server manually instead of using the application's own startup sequence.
4. Solution: The test script was refactored to use the app's start() and shutdown() methods. This ensured a clean, reliable test environment and confirmed that the API was fully functional.

Phase 5: Finalization and Repository Setup

• Goal: Prepare the project for a public release on GitHub.
• Process:
1. Updated the README.md to reflect the new project structure, features, and setup instructions.
2. Created a requirements.txt file to define the dependencies (pytest, requests) needed for testing.
3. Polished the .gitignore to exclude user-generated export files (ai_report.txt, file_dump_export.html).
4. Successfully initialized a local Git repository and pushed the final, polished code to a new remote repository on GitHub, resolving authentication issues with a Personal Access Token (PAT).

Conclusion

The project is now stable, well-structured, tested, and documented. The established pattern of a Python back-end serving a dynamic front-end is complete and ready to be replicated for other applications in the suite.