Software Requirements Specification (SRS) for Local File Manager

1. Introduction

1.1 Purpose

This document provides a detailed description of the requirements for the Local File Manager API. Its purpose is to define the system's features, capabilities, and constraints, serving as a foundational guide for development, testing, and stakeholder understanding. The system provides a secure RESTful API to manage files and directories within a designated local storage area on the server.

1.2 Document Conventions

This document uses Markdown for formatting. API endpoints are described using a standard format including the HTTP method and path.

1.3 Intended Audience

This document is intended for:

- **Software Developers:** To understand the system's architecture and implement the required features.
- **QA/Test Engineers:** To create test plans and test cases to verify system functionality.
- **Project Managers:** To understand the scope and requirements of the project.
- **System Administrators:** To understand the operating environment and deployment requirements.

1.4 Project Scope

The project is a backend REST API service that allows for remote management of a local file system directory. The scope includes:

- Listing the contents of directories.
- Creating, renaming, copying, moving, and deleting files and folders.
- Uploading files to a specified directory.
- Downloading a specified file.
- Creating a zip backup of the entire storage directory.
- Securing the file system by restricting all operations to a designated storage directory.

The project does **not** include a graphical user interface (GUI); it is a standalone API designed to be consumed by a separate client application (e.g., a web-based frontend).

1.5 References

- **Express.js:** Web application framework for Node.js.
- **Multer:** Node.js middleware for handling multipart/form-data, used for file uploads.
- **fs-extra:** Node.js module that adds file system methods not included in the native fs module.
- **Archiver:** Node.js module for creating archive files (e.g., zip).
- **OpenAPI Specification (Swagger):** For API documentation.

2. Overall Description

2.1 Product Perspective

The Local File Manager API is a self-contained, server-side application. It is designed to be the backend component of a client-server architecture. A separate frontend client (not included in this project) will make HTTP requests to this API to perform file management operations on behalf of a user. The system abstracts the server's file system, providing a secure and controlled interface for remote interaction.

2.2 Product Features

The major features of the system are:

- **Directory Browsing:** View the contents of any directory within the storage root.
- **File & Folder Manipulation:** Full CRUD (Create, Read, Update, Delete) operations for files and folders.
- **File Transfer:** Upload and download capabilities.
- **Path Validation:** A security middleware to prevent path traversal attacks and restrict access to the designated storage area.
- **Backup Creation:** On-demand creation of a zip archive of the entire storage directory.
- **API Documentation:** Self-documented API using Swagger/OpenAPI.

2.3 User Classes and Characteristics

The primary user of this API is a **Frontend Application**. The end-user interacts with the frontend, which in turn communicates with the API. The API assumes that the client application will handle user authentication and provide a user-friendly interface.

2.4 Operating Environment

- **OS:** The system is designed to run on any operating system that supports Node.js.
- **Runtime:** Node.js.
- **Dependencies:** Requires an internet connection for the client to access the API. No database is required.

2.5 Design and Implementation Constraints

- The system must be implemented using Node.js and Express.js.
- All file system operations must be strictly confined to the storage directory located at the project root. Any attempt to access paths outside of this directory must be blocked.
- The API must be stateless (RESTful).

2.6 Assumptions and Dependencies

- It is assumed that the server running the application has sufficient disk space for file storage and backup creation.
- The system depends on the Node.js runtime and the external npm packages listed in package.json.
- It is assumed that a separate client application will be developed to interact with this API.

3. System Features (Functional Requirements)

This section details the functional requirements of the system, broken down by feature.

3.1 Directory Browsing

- **3.1.1 Description:** The system shall allow a client to retrieve the list of all files and subdirectories within a specified directory.
- **3.1.2 API Endpoint:** GET /api/browse
- **3.1.3 Inputs:** A path query parameter representing the relative path of the directory to browse (e.g., . for the root, or folder1/subfolder).
- **3.1.4 Outputs:**
  - **Success:** A JSON array of file system items. Each item includes the name, isDirectory status, size in bytes, and lastModified timestamp.
  - **Error:** An error response if the path is invalid, not found, or outside the allowed storage directory.

3.2 File and Directory Creation

- **3.2.1 Description:** The system shall allow a client to create a new, empty file or a new directory at a specified path.
- **3.2.2 API Endpoints:**
  - POST /api/create-file
  - POST /api/create-folder
- **3.2.3 Inputs:** A JSON body containing a path property for the new file or folder (e.g., {"path": "new-folder/new-file.txt"}).
- **3.2.4 Outputs:**
  - **Success:** A JSON response confirming successful creation.
  - **Error:** An error response if the path is invalid or outside the allowed storage directory.

3.3 File and Directory Deletion

- **3.3.1 Description:** The system shall allow a client to delete an existing file or directory (including all its contents).
- **3.3.2 API Endpoint:** DELETE /api/delete
- **3.3.3 Inputs:** A JSON body containing a path property for the item to be deleted.
- **3.3.4 Outputs:**
  - **Success:** A JSON response confirming successful deletion.
  - **Error:** An error response if the path is invalid, not found, or outside the allowed storage directory.

3.4 File and Directory Renaming

- **3.4.1 Description:** The system shall allow a client to rename an existing file or directory.
- **3.4.2 API Endpoint:** PUT /api/rename
- **3.4.3 Inputs:** A JSON body containing oldPath and newPath properties.
- **3.4.4 Outputs:**
  - **Success:** A JSON response confirming the rename was successful.
  - **Error:** An error response if a path is invalid, not found, or outside the allowed storage directory.

3.5 File and Directory Copying

- **3.5.1 Description:** The system shall allow a client to copy a file or directory to a new destination.
- **3.5.2 API Endpoint:** POST /api/copy
- **3.5.3 Inputs:** A JSON body containing path (source) and destination properties.
- **3.5.4 Outputs:**
  - **Success:** A JSON response confirming the copy was successful.
  - **Error:** An error response if a path is invalid, not found, or outside the allowed storage directory.

3.6 File and Directory Moving

- **3.6.1 Description:** The system shall allow a client to move a file or directory to a new destination.
- **3.6.2 API Endpoint:** POST /api/move
- **3.6.3 Inputs:** A JSON body containing path (source) and destination properties.
- **3.6.4 Outputs:**
  - **Success:** A JSON response confirming the move was successful.
  - **Error:** An error response if a path is invalid, not found, or outside the allowed storage directory.

3.7 File Uploading

- **3.7.1 Description:** The system shall allow a client to upload one or more files to a specified directory.
- **3.7.2 API Endpoint:** POST /api/upload

- **3.7.3 Inputs:** A multipart/form-data request containing:
  - A path field for the destination directory.
  - One or more files fields containing the file data.
- **3.7.4 Outputs:**
  - **Success:** A JSON response confirming the number of files uploaded.
  - **Error:** An error response if the path is invalid or outside the allowed storage directory.

3.8 File Downloading

- **3.8.1 Description:** The system shall allow a client to download a single specified file.
- **3.8.2 API Endpoint:** GET /api/download
- **3.8.3 Inputs:** A path query parameter representing the relative path of the file to download.
- **3.8.4 Outputs:**
  - **Success:** The raw file data with Content-Disposition: attachment headers to trigger a browser download.
  - **Error:** An error response if the path is a directory, invalid, not found, or outside the allowed storage directory.

3.9 Storage Backup

- **3.9.1 Description:** The system shall create a zip archive of the entire storage directory and save it to the server's desktop.
- **3.9.2 API Endpoint:** POST /api/backup
- **3.9.3 Inputs:** None.
- **3.9.4 Outputs:**
  - **Success:** A JSON response confirming successful backup creation and providing the absolute path to the backup file.
  - **Error:** An error response if the backup process fails.

4. External Interface Requirements

4.1 User Interfaces

The system does not provide a direct user interface. It exposes a REST API that is documented via Swagger UI at the /api-docs endpoint.

4.2 Software Interfaces

The system provides a RESTful API for programmatic access. Client applications will interact with the system by making HTTP requests to the available endpoints. The API uses standard HTTP methods (GET, POST, PUT, DELETE) and returns data in JSON format.

4.3 Hardware Interfaces

No specific hardware interfaces are required beyond what is necessary to run a standard Node.js application.

4.4 Communications Interfaces

- The API is accessible via the HTTP/S protocol.
- A TCP/IP network connection is required for a client to communicate with the server.

5. Non-Functional Requirements

5.1 Performance Requirements

- The API should respond to most requests within 500ms under normal load.
- File upload and download speeds are dependent on network bandwidth and server disk I/O, but the application should not introduce significant overhead.

5.2 Safety Requirements

There are no specific safety requirements as the system does not control any life-critical or hazardous equipment.

5.3 Security Requirements

- **Path Traversal:** The system must prevent any and all attempts at path traversal. All file system access must be strictly limited to the storage directory and its subdirectories.
- **Error Handling:** The system must handle errors gracefully and provide clear, structured error messages without exposing sensitive system information (e.g., full file system paths, stack traces) in production responses.
- **CORS:** For production deployment, the Cross-Origin Resource Sharing (CORS) policy should be configured to a specific whitelist of trusted frontend domains rather than the * wildcard.

5.4 Software Quality Attributes

- **Reliability:** The server should be stable and handle concurrent requests without crashing.
- **Maintainability:** The code should be well-structured, commented where necessary, and easy to understand and modify.
- **Scalability:** While designed for single-server deployment, the use of DiskStorage for uploads ensures it can handle large files without consuming excessive memory.