File Wizard

A Scalable File Management Application

Gabriel Perez
GabrielPerezCSdev@gmail.com

Contents

File Wizard

The **File Wizard** application is a desktop utility designed to streamline file system management through an intuitive user interface and powerful backend operations. It combines modern web technologies with a robust backend to deliver seamless and efficient functionality.

Overview

File Wizard simplifies file organization, search, and metadata management for users, leveraging a modular architecture that separates concerns between the frontend and backend. It is designed to be scalable, maintainable, and extendable for future enhancements.

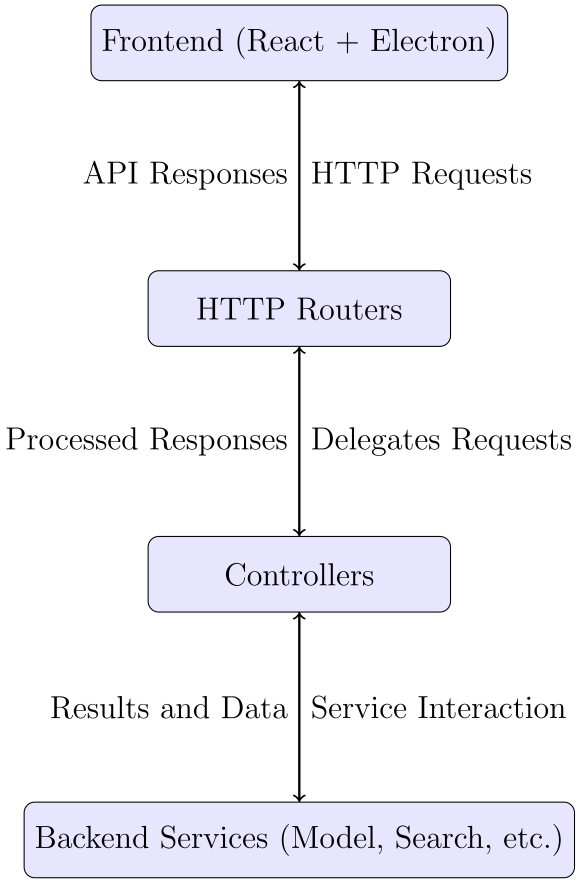
Technology Stack

File Wizard leverages a modern technology stack, employing powerful tools and languages for performance, scalability, and maintainability:

- **Frontend:**
  - **Languages:** JavaScript (ES6+), TypeScript.
  - **Libraries and Frameworks:** React (for UI components), Electron (for desktop application support).
  - **Build Tools:** Webpack (for module bundling), npm (for dependency management).
- **Backend:**
  - **Language:** Rust.
  - **Concurrency:** Uses Rust's `std::thread`, `Arc`, and `Mutex` for thread-safe operations and efficient parallelism.
- **Communication:**
  - **Language:** Rust.
  - **Protocol:** HTTP.
  - **Libraries:** Actix Web (for HTTP routers), Serde (for JSON serialization/deserialization).

Architecture Schema

The diagram below represents the overall architecture of File Wizard, illustrating the bidirectional interaction between its components.



Key Highlights

- **Frontend-Backend Interaction:** The React-based frontend communicates with the backend exclusively through HTTP requests to routers, ensuring a clean separation of concerns and scalable integration.
- **Router Layer:** The routers, implemented in Rust, handle HTTP requests from the frontend. These routers delegate the requests to controllers for further processing.
- **Controller Layer:** Controllers process requests from the routers and interact with backend services to execute application logic.
- **Backend Services:** The backend folder hosts core services, which implement essential functionality such as file system management, search operations, and metadata handling.
- **Scalable Frontend Architecture:** Built with React and Electron, the frontend UI is modular and designed to support responsiveness and the seamless addition of new features.
- **Thread-Safe Design:** Rust's `Arc` and `Mutex` ensure robust handling of concurrent operations within the backend services, enabling safe parallel execution.

Backend Design

Model

The **Model** folder is a core part of the File Wizard backend, responsible for representing and handling files, folders, and their associated metadata. This design ensures thread safety, modularity, and extensibility for managing a file system structure.

File.rs

The `File.rs` module defines the `File` struct, which represents individual files in the file system. It includes methods for retrieving file metadata and interacting with parent folders. Key features include:

- **Thread Safety:** The `parent` attribute uses `Arc<Mutex<Folder>>` for shared, thread-safe access to parent folders.
- **Metadata Management:** Uses a `HashMap` to store metadata such as size, creation date, and file-specific attributes (e.g., file extension).
- **Helper Methods:**
  - `new`: Constructs a `File` instance from a given `Path`.
  - `get_metadata`: Returns a reference to the metadata.
  - `get_raw_size`: Retrieves the file's size from the metadata.

Folder.rs

The `Folder.rs` module defines the `Folder` struct, representing directories in the file system. This module supports hierarchical relationships and metadata aggregation for child elements. Key features include:

- **Hierarchical Structure:** Tracks child files and folders using a `Vec<PathType>`.
- **Thread-Safe Mutability:** Uses `Arc<Mutex<Folder>>` for safe updates to parent references.
- **Metadata Aggregation:** Automatically updates size and metadata based on child elements.
- **Helper Methods:**
  - `new`: Constructs a `Folder` instance from a given `Path`.
  - `add_child`: Adds a child `PathType` (file or folder) and propagates metadata updates.
  - `update_size`: Updates folder size and propagates changes up the hierarchy.

Metadata.rs

The `Metadata.rs` module provides utility functions to extract and manage metadata for files and folders. It handles operations like size formatting and timestamps. Key features include:

- **Unified Metadata Extraction:** Functions like `file_folder_metadata` ensure consistent metadata handling for both files and folders.
- **Platform-Specific Access Control:** Includes functions to check accessibility (e.g., read permissions).
- **Human-Readable Formats:** Converts raw sizes into readable strings (e.g., KB, MB).

PathType.rs

The `PathType.rs` module defines the `PathType` enum to unify the representation of files and folders. Key features include:

- **Enum Variants:**
  - `File`: Represents a file using `Arc<Mutex<File>>`.
  - `Folder`: Represents a folder using `Arc<Mutex<Folder>>`.
  - `None`: Represents an invalid or inaccessible path.
- **Display Implementation:** Implements the `fmt::Display` trait for human-readable descriptions of files and folders.

The **Model** folder ensures modular handling of file system entities while maintaining thread-safe operations and clear hierarchy management.

Search

The **Search** folder implements the core logic and utilities for exploring and managing file system searches. It leverages modular design to maintain clarity, thread-safety, and extensibility.

PathMap.rs

The `PathMap.rs` module defines the `PathMap` structure, a custom data structure for managing mappings between URLs and their corresponding entities (`File` and `Folder`). Key features include:

- **Folder and File Tracking:** Manages two `HashMap` instances for efficient retrieval of files and folders by their URLs.
- **Thread-Safe Ownership:** Uses `Arc<Mutex<Folder>>/Arc<Mutex<File>>` for shared ownership and safe concurrent access.
- **Operations:**
  - `add`: Adds a `PathType` (either file or folder) to the appropriate map.

- `get`, `get_folder`, `get_file`: Retrieve entries by URL.
- `contains`: Checks if a URL exists in the maps.
- `remove`: Removes an entry and returns it as a `PathType`.

ThreadManager.rs

The `ThreadManager.rs` module provides utilities for managing the lifecycle of search threads. It ensures seamless thread creation, state transitions, and termination. Key components include:

- **Thread States:** Uses the `State` enum to represent the thread's status (`RunningInit`, `Paused`, `Stopped`, etc.).
- **Thread Lifecycle Management:**
  - `spawn_thread`: Spawns a new thread to execute search operations.
  - `pause_thread`, `resume_thread`, `stop_thread`: Manage thread states dynamically.
- **Safe Multithreading:** Shares access to thread state and search operations using `Arc<Mutex>` for synchronization.

Utils.rs

The `Utils.rs` module implements the main `Search` struct and related search logic. It orchestrates the folder and file traversal process and integrates with `PathMap` and thread management. Key features include:

- **Search Initialization:**
  - `initialize_search`: Initializes the search by setting the root directory, creating entries in `PathMap`, and discovering children.
  - `set_root_search_directory`: Validates and normalizes the root search directory path.
- **Search Execution:**
  - `execute_search`: Executes the search process, managing the `frontier_map` and assigning the next directory to `current_dir`.
  - `discover_immediate_children`: Traverses the current directory to discover and classify child entries as files or folders.
- **Frontier Management:**
  - `sort_frontier_list`: Sorts the frontier list based on future heuristics.
  - `pop_frontier_entry`: Removes and returns the next folder to process.
- **Path Management:** Leverages `PathMap` for adding and retrieving file and folder paths during traversal.

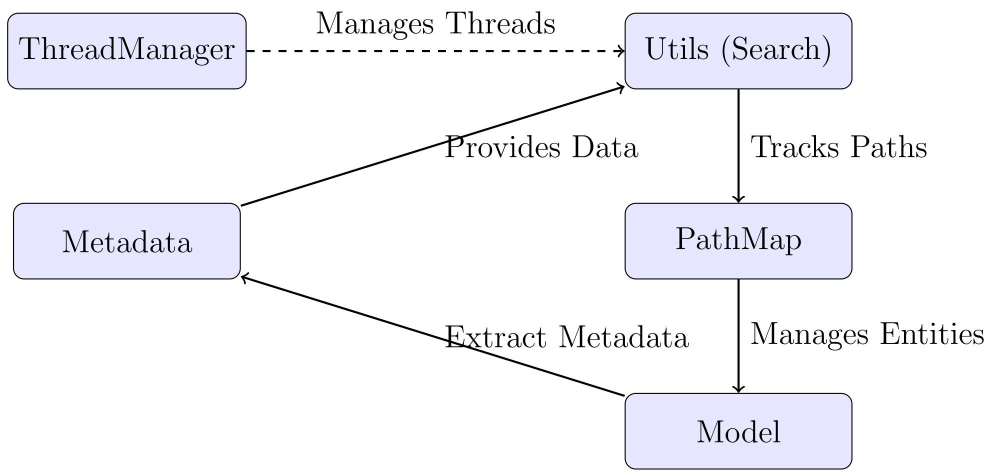
Relationships Between Components

- `Utils.rs` acts as the orchestrator for search logic, integrating with:
  - `PathMap.rs` for path tracking.
  - `ThreadManager.rs` for thread management during long-running operations.
  - The `Model` folder (`File`, `Folder`, `PathType`) for file system entities.
- `PathMap.rs` serves as the storage and retrieval backbone for search operations.
- `ThreadManager.rs` ensures that search operations run in a non-blocking, thread-safe manner.

The **Search** folder enables modular and efficient search processes while maintaining clear separation of concerns between thread management, path tracking, and search logic.

Search Schema

The following diagram illustrates the interaction between the **Search** components, highlighting the flow of metadata, thread management, and path tracking.



Key Points:

- **Utils (Search)**: Orchestrates search operations, integrating with `PathMap`, `ThreadManager`, and `Model`.
- **PathMap**: Tracks and retrieves file and folder paths for search traversal.
- **ThreadManager**: Manages thread states (e.g., running, paused) for non-blocking operations.
- **Model**: Represents file and folder structures, providing metadata and hierarchical relationships.
- **Metadata**: Extracted from file and folder entities, used to support search operations.

Controllers

The **Controllers** section details the interface layer connecting backend components to the user or higher-level application logic. Each controller is responsible for managing a specific aspect of the backend functionality. Unlike the **Search** module, the controllers do not depend on enums or hierarchical structures but rather act as standalone modules, each tightly coupled with the backend components they serve.

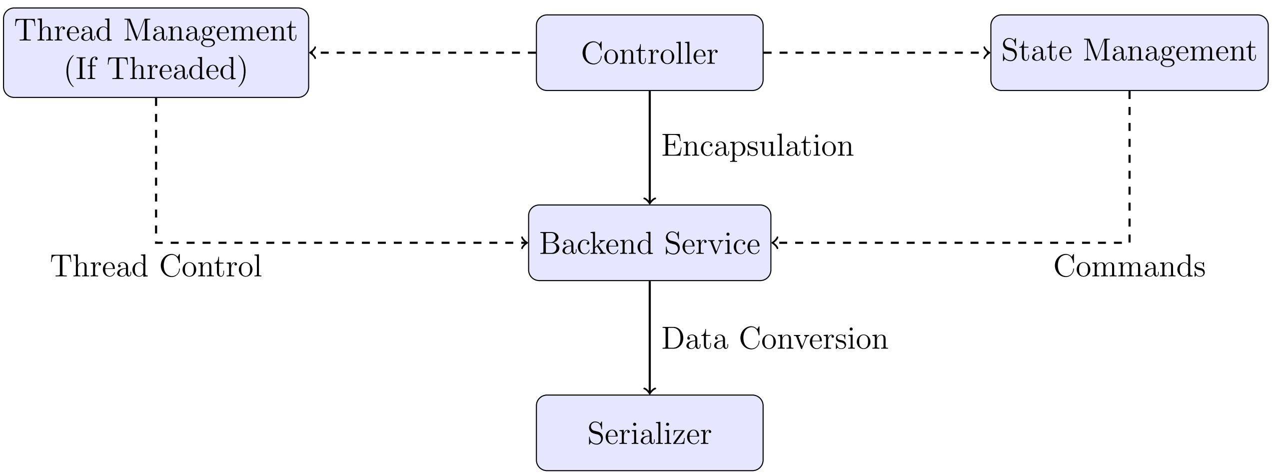
General Overview

Controllers follow a consistent design schema:

- **Encapsulation of Backend Logic:** Each controller encapsulates backend functionality, exposing methods for interacting with underlying services (e.g., search, serialization).
- **Thread-Safe Operations:** Use of `Arc<Mutex<T>>` ensures safe concurrent access to backend components, maintaining synchronization across threads.
- **State Management:** Controllers often manage the state of their associated backend service, allowing operations such as starting, pausing, resuming, and stopping processes.
- **Data Serialization:** Implement serialization of complex data structures into formats suitable for communication with external interfaces, such as JSON.
- **Modular Interaction:** Designed to be modular, each controller can be extended or modified without affecting other controllers.

Controller Design Schema

The following diagram represents the typical interaction between a controller, its backend service, and serialization components:



Key Points:

- Controllers wrap backend logic, exposing high-level methods to external interfaces.

- Shared states (e.g., thread states, data models) are managed using thread-safe abstractions such as `Arc` and `Mutex`.
- The **Serializer** module converts complex Rust data structures into JSON, facilitating communication with the frontend.
- Modular design allows for the addition of new controllers and serialization mechanisms without impacting existing functionality.

Serialization Module

The **Serializer** module plays a crucial role in bridging the backend services and the external interfaces by converting complex Rust data structures into JSON format. This module is especially important for enabling the frontend application to interpret and display data received from the backend.

Purpose and Functionality

The Serializer module is responsible for:

- **Data Conversion:** Transforming backend entities such as `Folder`, `File`, and `PathType` into serializable formats.
- **JSON Serialization:** Utilizing the `serde` library to serialize data structures into JSON strings.
- **Maintaining Data Integrity:** Ensuring that all relevant metadata and hierarchical relationships are preserved during serialization.

Implementation Details

The Serializer module includes:

- `SerializableFolder`: A struct that represents a serializable version of the `Folder` entity, including fields like `name`, `url`, `children`, `metadata`, `index`, and `num_children`.
- `SerializableFile`: A struct for serializing the `File` entity, capturing essential attributes like `name`, `url`, and `metadata`.
- `SerializablePathType`: An enum that differentiates between serializable files and folders, enabling the representation of the `PathType` enum in JSON.
- **Conversion Implementations**: `From` trait implementations that facilitate the conversion from backend entities to their serializable counterparts.

Integration with Controllers

The Serializer module is integrated into controllers as follows:

- Controllers invoke the Serializer to convert backend data into JSON before sending responses to the frontend.

- This separation ensures that serialization logic is centralized, promoting code reuse and maintainability.
- By handling serialization within the controllers, the application maintains a clear separation between data processing and presentation layers.

Interaction with Backend Components

The controllers interact closely with backend services, adhering to the following principles:

- **Encapsulation:** Backend components such as models or search utilities are never directly exposed to the user or higher-level modules; controllers provide controlled access.
- **Synchronization:** Use of shared ownership (`Arc`) and locking mechanisms (`Mutex`) ensures safe concurrent modifications.
- **Data Serialization:** The Serializer module converts complex data structures into JSON, enabling controllers to communicate data to external interfaces efficiently.
- **Extensibility:** Each controller is designed to be self-contained, allowing new controllers and serialization methods to be added without significant changes to the existing architecture.

Modularity and Scalability

By maintaining loose coupling and strong encapsulation, the controller layer supports the following:

- **Scalability:** Adding new backend components requires only the creation of corresponding controllers and serialization mappings.
- **Maintenance:** Controllers abstract the backend logic, isolating changes to a single layer. The Serializer module centralizes data conversion logic, simplifying updates.
- **Flexibility:** Controllers and the Serializer can expose additional features or adapt existing ones without interfering with other modules.

Ports

The **Ports** layer serves as the entry point for external interactions with the application, handling communication through various protocols. This layer defines and manages components like HTTP routers and will eventually include other mechanisms like WebSockets or message queues.

Routers

The **Routers** subsection focuses on the HTTP-based routing components of the `Ports` layer. Routers define endpoints for external clients, map requests to backend controllers, and encapsulate logic for handling incoming traffic.

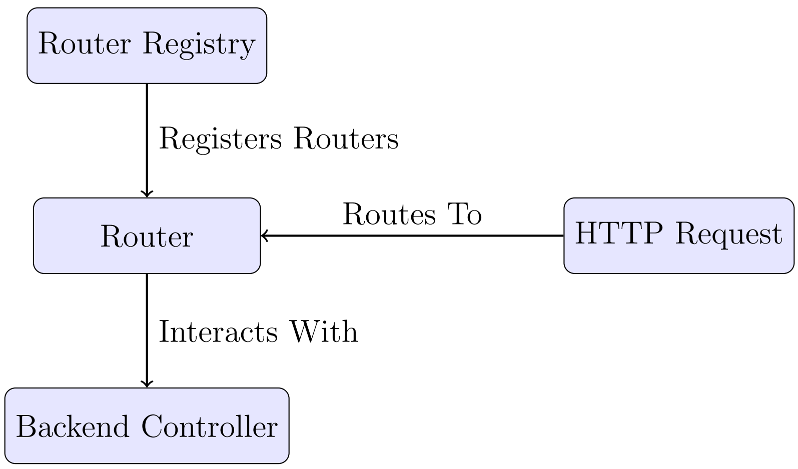
General Overview

Routers follow a standardized design schema:

- **Dynamic Routing:** Routers define and handle HTTP endpoints that interact with specific backend controllers.
- **Trait-Based Abstraction:** The `Router` trait ensures a consistent interface for all routers, with methods for initialization and request handling.
- **Concurrent Operations:** Leveraging `async` functions and the `RouterRegistry`, routers can initialize and handle multiple requests concurrently.
- **Extensibility:** The `RouterRegistry` allows for easy addition of new routers, enabling modular growth of the application's routing layer.

Router Design Schema

The following diagram illustrates the interaction between the `RouterRegistry`, individual routers, and backend controllers:



Key Points:

- The `RouterRegistry` serves as a central hub for registering and managing routers.
- Routers map HTTP requests to backend controllers, ensuring efficient and modular handling of external interactions.
- Each router defines endpoints and the logic to handle requests, often leveraging backend controllers for business logic.

Interaction with Backend Components

Routers interact with backend controllers through the following principles:

- **Decoupling:** Routers abstract backend functionality, exposing only the endpoints required for client interaction.
- **Thread Safety:** Use of `Arc<Mutex<T>>` ensures safe, concurrent access to backend controllers.
- **Modularity:** Each router is dedicated to a specific backend functionality, such as search or file management, allowing independent development and testing.

Modularity and Scalability

The **Routers** subsection is designed for scalability and ease of maintenance:

- **Extensibility:** New routers can be added by implementing the `Router` trait and registering them with the `RouterRegistry`.
- **Concurrency:** Asynchronous design ensures routers can handle multiple requests simultaneously.
- **Adaptability:** The centralized `RouterRegistry` allows for easy configuration and management of all application routers.

Frontend Design

The **Frontend Design** section outlines the architecture and structure of the File Wizard application's user interface. Built with **Electron** and **React**, the frontend leverages modern web technologies to deliver a seamless, interactive desktop experience.

General Overview

The frontend architecture is designed to balance modularity, responsiveness, and interactivity, adhering to the following principles:

- **Component-Based Architecture:** Utilizes React's component model to build a modular and reusable UI.
- **Separation of Concerns:** Divides the application into components, controllers, and utilities to maintain a clean and maintainable codebase.
- **Routing and Navigation:** Implements client-side routing using `react-router-dom` for seamless navigation between pages.
- **State Management:** Manages state using React hooks (`useState`, `useEffect`) to ensure responsive and interactive UI elements.
- **Backend Integration:** Communicates with backend services via HTTP requests, encapsulated within controller modules to separate business logic from UI components.
- **Scalability:** Designed to support the addition of new features and pages with minimal refactoring, promoting long-term maintainability.

Application Flow

The following diagram illustrates the flow of data and interactions within the frontend application:

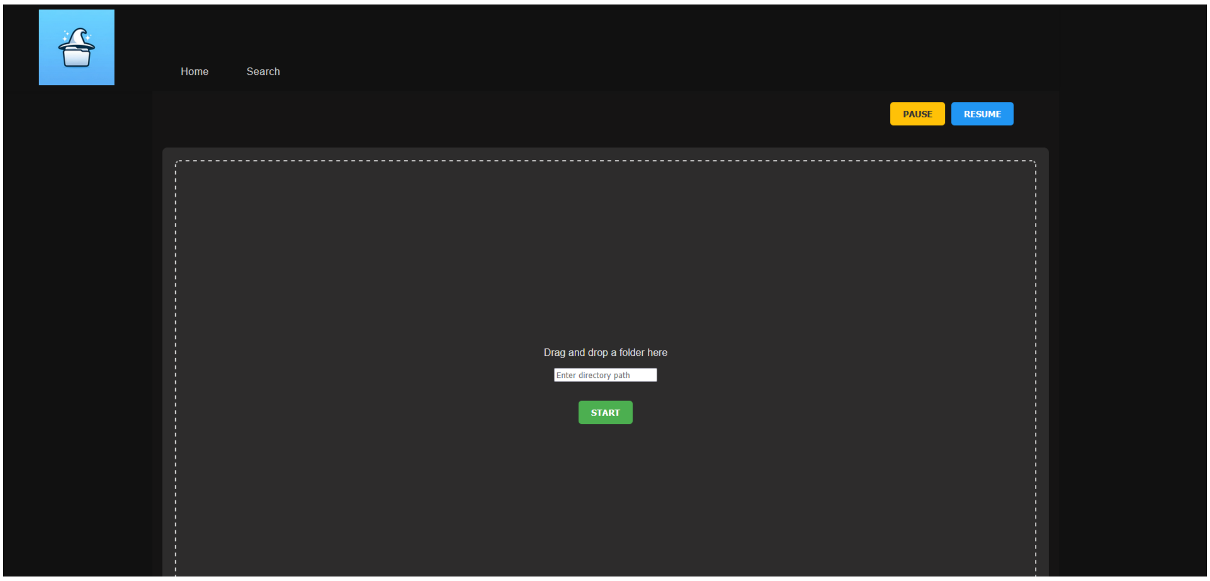
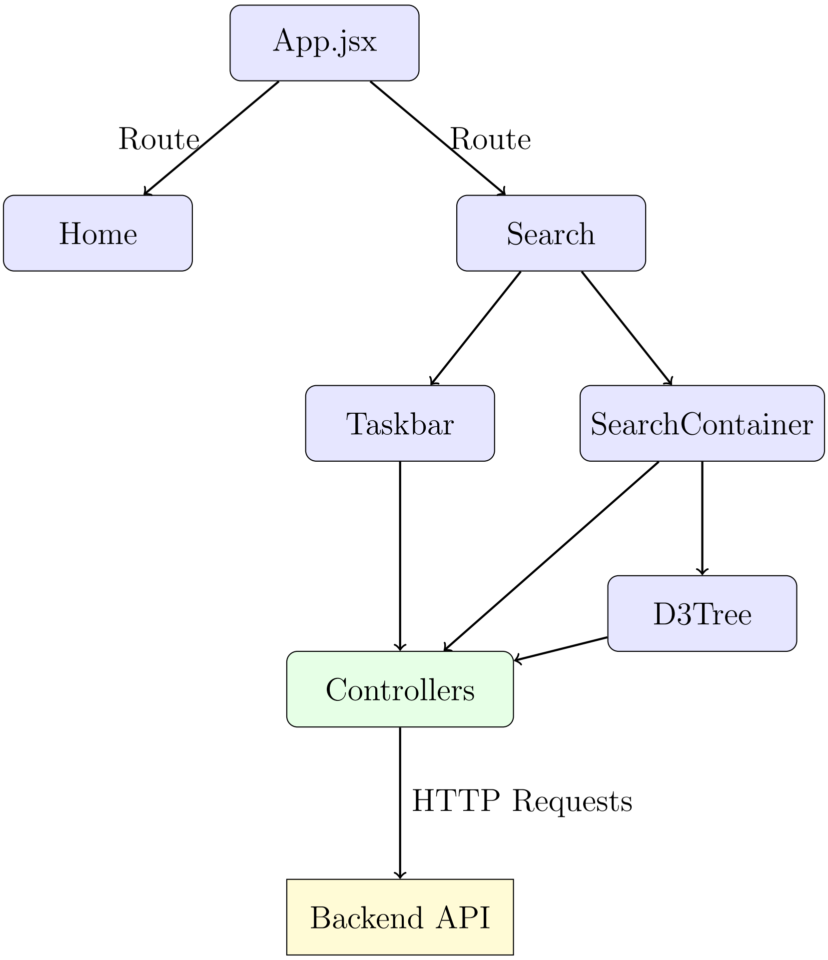


Figure 1: File Wizard home page showing the initialization view.



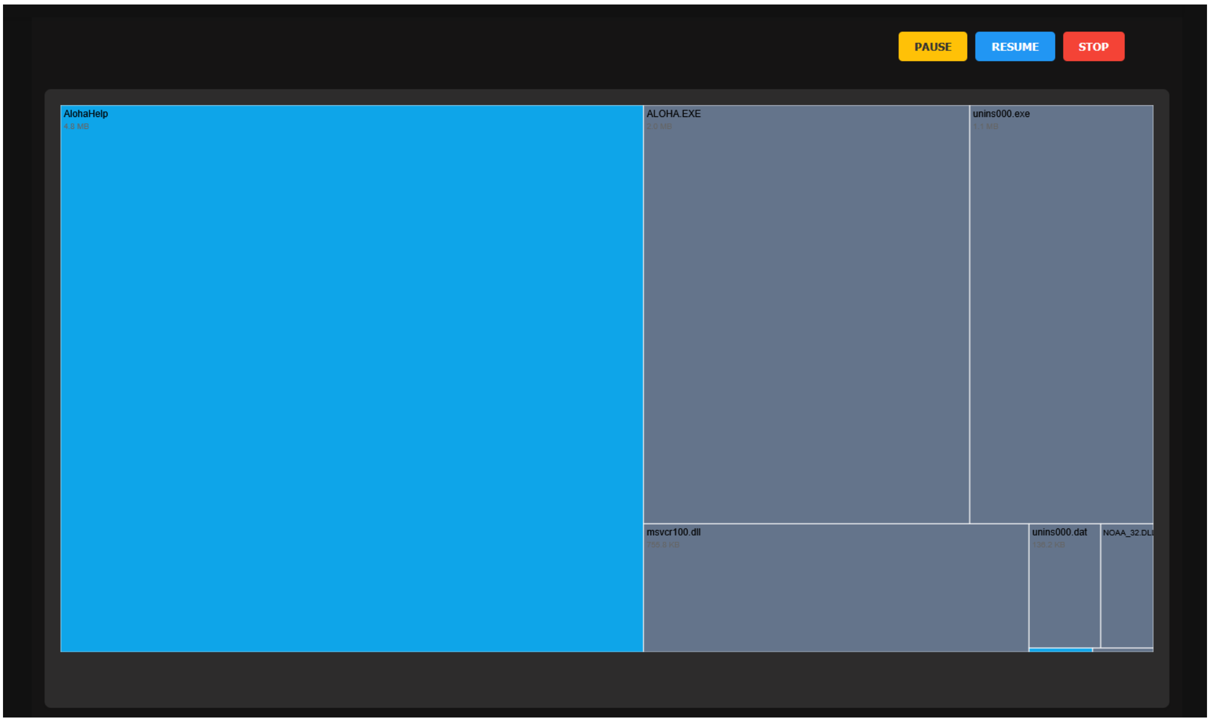


Figure 2: Search initialized with folder selection and dynamic visualization.

Displaying Results

Once a search is initiated, the user can observe the dynamic updates of file sizes and visualizations. The D3.js-based `D3Tree` component renders the file system structure in real time, allowing the user to explore the directory contents.

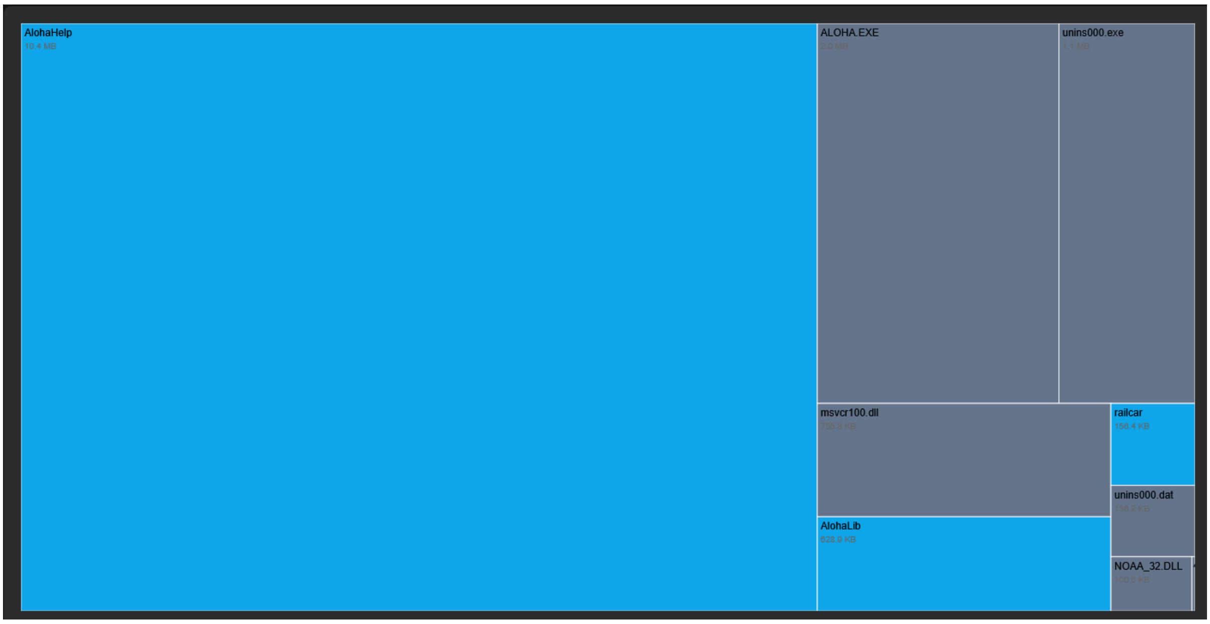


Figure 3: Dynamic updates showing updated file sizes during the search.

Interacting with the Visualization

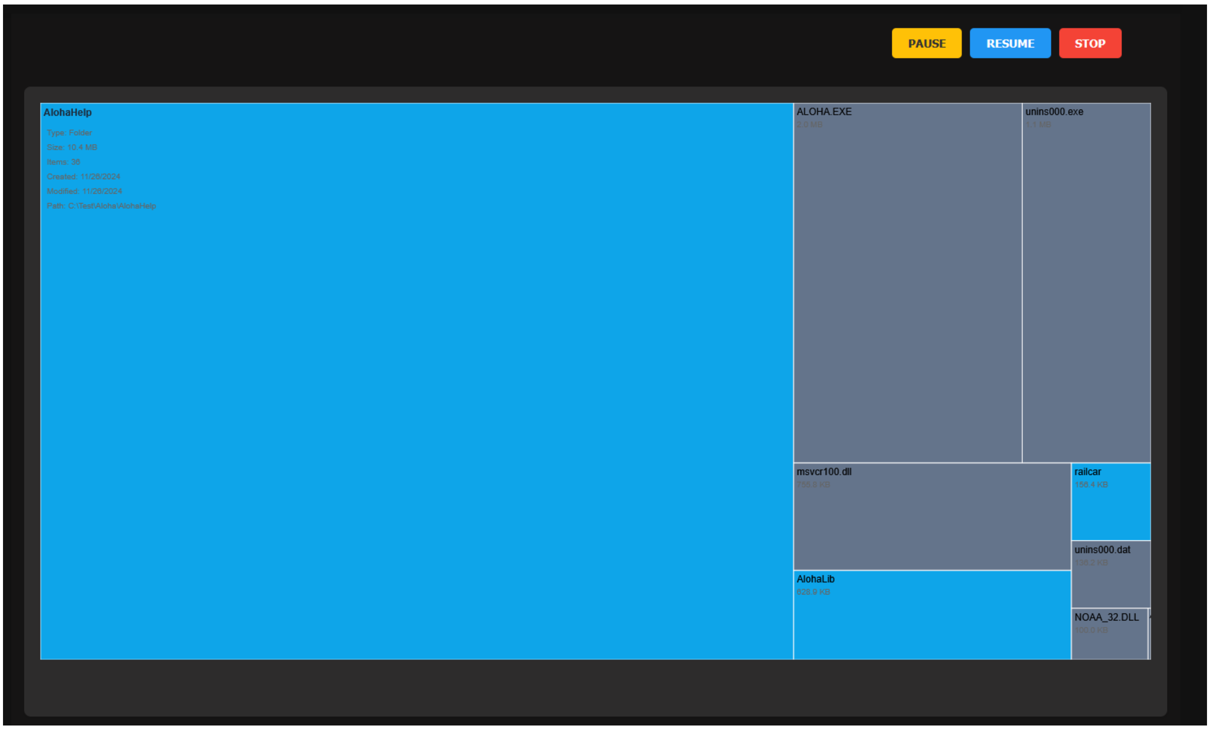


Figure 4: Folder metadata displayed during navigation in the interactive visualization.

Users can interact with the `D3Tree` visualization by clicking on folders to navigate deeper into the file system hierarchy or simply hovering over a file or a folder to view the metadata. The application efficiently updates the view to reflect the selected folder's contents.

Conclusion

The frontend of File Wizard is built to provide an intuitive and interactive user interface for managing and visualizing the file system. By leveraging React and D3.js within an Electron application, it offers a modern desktop experience with efficient performance and scalability. The architecture emphasizes separation of concerns, allowing for easy maintenance and future enhancements.

Figure 5: Traversal to a new folder via the interactive visualization.