Here is the functional, architectural, and structural blueprint for your Electron-React File Manager.

Architecture and Structure

Your application will follow the standard Electron Dual-Process Architecture with a clear separation of concerns, which is essential for performance and security in a file management tool.

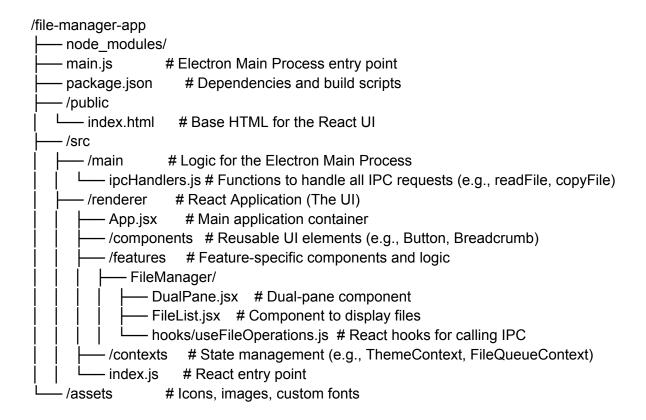
1. Architectural Blueprint

Component	Responsibility	Technologies	
Main Process (Node.js)	The application's "brain." Manages the window, system-level tasks, and all privileged I/O (Input/Output) like file system access.	Node.js, fs, path, ipcMain, electron-builder	
Renderer Process (UI)	The application's "view." Renders the user interface and handles user interactions (clicks, key presses). It never touches the file system directly.	React, JavaScript/TypeScript, HTML/CSS (Tailwind), ipcRenderer	
Inter-Process Communication (IPC)	The communication bridge between the Renderer and Main processes. Used to safely request file operations.	ipcRenderer.invoke() / ipcMain.handle()	
Data Persistence	Storage for user-specific settings and application-specific metadata.	electron-store (for settings), SQLite/PouchDB (for tags/favorites)	

2. Project Structure

A well-structured project is critical for an Electron-React app:





Core Functionalities and Activity Flows

1. Primary Activity Flow: Directory Navigation (Read Operation)

This is the most frequent activity and defines the app's responsiveness.

| Step | User Action / Component | Process | Description |

```
|:---|:---|
```

- | 1. Interaction | User clicks on a folder name in <FileList /> or a link in the <AddressBar />. | Renderer | Triggers a React function to change the current path state. |
- | 2. IPC Request | The React component calls a hook: useFileOperations().readDirectory(newPath). | Renderer \$\to\$ Main (via ipcRenderer.invoke) | Sends the requested path string to the Main Process. |
- | 3. File Access | ipcMain handler in ipcHandlers.js receives the path. | Main (Node.js) | Executes fs.readdir(newPath, { withFileTypes: true }) to get file data. |
- | 4. Response | The Main Process returns a sanitized array of file/folder objects. | Main \$\to\$ Renderer | Sends data back (e.g., [{name: 'docs', isDirectory: true}, ...]). |
- | 5. UI Update | The React hook receives the data and updates the app's central React state (currentFiles). | Renderer | <FileList /> re-renders instantly with the new directory contents. |

2. Secondary Activity Flow: File Copy/Move (Write Operation)

This involves a longer-running task that should be handled asynchronously.

| Step | User Action / Component | Process | Description |

- 1. Interaction | User initiates a copy/move (e.g., drag and drop, or \$\text{Cmd}+\text{X}, \text{Cmd}+\text{V}\$). | Renderer | Collects source paths and the single destination path. |
- 2. IPC Request | Component calls a command: useFileOperations().startTransfer({source, destination, type}). | Renderer \$\to\$ Main (via ipcRenderer.send) | Sends the operation details to the Main Process. Note: Using send because no immediate return value is needed.
- 3. Background Task | ipcMain handler starts the file operation. | Main (Node.js) | Uses the Node.js fs module or a library like ncp for recursive copies. Important: The Main Process runs this in the background. |
- 4. Progress Updates | During the operation, the Main Process periodically emits progress data (e.g., 50% complete, 4.5GB/9GB). | Main \$\to\$ Renderer (via mainWindow.webContents.send()) | Sends status updates to the UI. |
- | 5. Queue Display | A React component like < Transfer Queue /> listens for progress updates and updates the progress bar/UI in real-time. | Renderer | User sees a detailed, live progress bar, just like in Windows Explorer. |



🕏 React Component Structure

The UI should be built with maximum modularity to support the dual-pane feature and state management.

1. Main Components

Component	Responsibility	Key Props/State
<app></app>	Main layout, state provider, and global hotkey listener (e.g., \$\text{Cmd}+\text{Shift}+\text{Dot}\$ for hidden files).	ThemeContext, Hotkeys
<dualpanelayout></dualpanelayout>	Container for two separate file view components, managing their independent	leftPath, rightPath

	paths and synchronized drag-and-drop actions.	
<pre><fileexplorerpane></fileexplorerpane></pre>	A single instance of the file viewer. Contains all sub-components for one directory.	currentPath, viewMode, onPathChange
<addressbar></addressbar>	Displays the full, editable path (typeable).	currentPath, onPathEdit
<filelist></filelist>	Displays the actual files and folders (List View, Icon View, or Column View). Handles selection and context menus.	files, sortBy, onFileClick
<sidebar></sidebar>	Displays Favorites (from SQLite), connected drives, and common folders (Desktop, Downloads).	favorites (from persistence)
<transferqueue></transferqueue>	The persistent overlay/side panel showing active and historical file operations.	activeTransfers, transferHistory

2. Essential React Hooks

Hook	Purpose	Calls IPC?
useFileOperations()	Primary hook for all file system interactions (read, copy, delete, rename).	YES
usePathState()	Manages the current path and path history (forward/back buttons).	NO

useGlobalHotkeys()	Captures keyboard shortcuts (e.g., \$\text{Cmd}+\text{K}\$ for "Go to Server," or \$\text{Delete}\$ to move to trash).	YES (for actions)
useAppSetting(key)	Manages application-wide settings like theme, default view mode, and hidden files toggle.	YES (reads/writes to electron-store)

