MediaSphere

OPEN SOURCE COMMUNITY

Official Documentation: MediaSphere PDF Viewer

Version 1.0 | **Last Updated:** July 2, 2025

Abstract

The MediaSphere PDF Viewer is a cross-platform desktop application engineered for efficient PDF document rendering and management. As a component of the open-source MediaSphere Suite, it is developed using Electron.js to ensure performance, modularity, and a consistent user experience. The application features a robust rendering engine based on Mozilla's PDF.js, an integrated file library, and a secure architecture that separates the main process from the renderer process. This document provides a comprehensive overview of the project's architecture, component specifications, installation procedures, and future development roadmap.

1.0 INTRODUCTION

The MediaSphere PDF Viewer is a standalone application designed for high-fidelity viewing of Portable Document Format (PDF) files. It is a foundational component of the MediaSphere Suite, a broader open-source initiative aimed at creating a unified platform for various media formats. The development is a collaborative effort, with significant contributions from the open-source community and members of GITAM (Deemed to be University).

The primary objective of the MediaSphere Suite is to consolidate disparate media applications into a single, cohesive framework. The suite currently comprises several active projects:

- **PDF Viewer:** MediaSpherePDFViewer
- **EPUB Viewer:** MediaSphereEPUBViewer
- **Document Viewer (.txt, .docx):** MediaSphereDocs
- **Music Player:** MediaSphereMusicPlayer
- **Video Player:** MediaSphereVideoPlayer

1.1 Key Features

The application provides the following core functionalities:

- **Interactive Document Navigation:** Allows users to move between pages sequentially using "Next" and "Previous" buttons or jump directly to a specific page number.
- **Dynamic Zoom Control:** Provides magnification controls to zoom in and out, allowing for detailed inspection of document content.
- **Integrated File Library:** A persistent sidebar that enables users to browse, search, and manage their local PDF files without leaving the application.

- **High-Fidelity Rendering Engine:** Utilizes the robust PDF.js library to ensure that documents are displayed with accurate colors, fonts, and layouts.
- **Text Layer Support:** Renders a transparent text layer over the document image, enabling standard text selection, copying, and search operations.
- **Ergonomic Theming:** Features a dark-themed user interface designed to reduce ocular strain during extended reading sessions.

2.0 INSTALLATION AND EXECUTION

2.1 Prerequisites

A working installation of Node.js is required to run the application.

2.2 Procedure

Execute the following commands in a terminal or command prompt:

1. **Clone the source repository:**
   git clone https://github.com/AtheeqAhmedMJ/MediaSpherePDFViewer.git
2. **Navigate to the project directory:**
   cd MediaSpherePDFViewer
3. **Install dependencies:**
   npm install
4. **Execute the application:**
   npm start

3.0 CONTRIBUTION GUIDELINES

Contributions to the project are encouraged. The development process prioritizes performance, user experience, and maintainable code.

3.1 Contribution Workflow

1. **Fork** the main repository.
2. Create a new, descriptive branch for the feature or bugfix (`git checkout -b feature/your-feature-name`).
3. Commit changes with clear, descriptive messages.
4. Push the branch to the forked repository.
5. Submit a **Pull Request** to the main repository with a detailed explanation of the changes

4.0 SYSTEM ARCHITECTURE

The application is built using the **Electron.js** framework. This choice allows us to use web technologies (HTML, CSS, JavaScript) to create a desktop application that works on Windows,

macOS, and Linux. The architecture is fundamentally divided into two distinct processes to enhance security and stability.

1. **Main Process:** This is the application's backend, running in a Node.js environment. It has full access to the computer's operating system and is responsible for creating windows and handling native interactions like opening file dialogs. The entry point for this process is `main.js`.
2. **Renderer Process:** This is the application's frontend, the user interface that runs inside a sandboxed Chromium browser window. It is responsible for displaying the HTML and CSS and executing the user-facing JavaScript (`renderer.js`). For security, this process is strictly forbidden from directly accessing the operating system.

Communication between these processes is arbitrated by a **Preload Script (`preload.js`)**, which acts as a secure bridge, exposing specific backend functionalities to the frontend in a controlled manner.

4.1 Data Flow Example: Opening a File

To understand the architecture, consider the flow of events when a user clicks "Open File":

1. **User Action:** The user clicks the "Open File" button in the Renderer Process (the UI).
2. **Secure API Call:** The UI's JavaScript (`renderer.js`) calls `window.electronAPI.openFile()`. This function is made available by the Preload Script.
3. **IPC Message:** The Preload Script sends a secure message (`'open-file-dialog'`) to the Main Process.
4. **Native Action:** The Main Process (`main.js`) receives the message and executes the code to open the operating system's native file selection dialog.
5. **File Read:** Once the user selects a file, the Main Process reads its data from the disk.
6. **IPC Response:** The Main Process sends the file data back to the Renderer Process in a new message (`'file-opened'`).
7. **UI Update:** The Renderer Process receives the file data and uses the PDF.js library to render the document on the screen.

5.0 COMPONENT SPECIFICATION

This section details the function and design rationale of each core file in the project.

5.1 Project Manifest (`package.json`)

This file serves as the project's configuration and dependency manifest. It is essential for managing the project's metadata and the external libraries it uses.

- `"main": "main.js"`: This entry is critical. It tells Electron where the application's Main Process script is located, making it the starting point of execution.
- `"scripts": { "start": "electron ." }`: This defines a convenient shortcut. Instead of typing `electron .` every time, developers can simply run `npm start` to launch the application.
- `"devDependencies": { "electron": "..." }`: This lists packages required for development. Declaring Electron here allows `npm` to download and install the framework needed to build and run the application.

5.2 Main Process (`main.js`)

This script is the application's controller, managing backend logic and OS-level interactions.

- **Function: `createWindow()`**
  - **Situation:** An application requires a graphical window for user interaction. Without it, the user would have nothing to see or click on.
  - **Task:** Define the properties of the main application window (like its size) and instantiate it.
  - **Action:** A `BrowserWindow` instance is created. The `webPreferences.preload` option is configured to load `preload.js`. This step is crucial for security, as it injects our secure bridge script before any other web content, ensuring the communication channel is established safely.
  - **Result:** A native desktop window is created, prepared to load the `index.html` user interface. This window acts as the container for our application's entire visual component.
- **IPC Handler: `ipcMain.on('open-file-dialog', ...)`**
  - **Situation:** The UI needs to allow users to select files from their computer. However, letting frontend JavaScript access the entire file system is a major security risk.
  - **Task:** Implement a secure, isolated service in the Main Process that the UI can request to open a file dialog on its behalf.
  - **Action:** An Inter-Process Communication (IPC) listener is established using `ipcMain.on()`. When a `'open-file-dialog'` message is received from the renderer, this handler executes Electron's `dialog` module to display a native file selection window. Upon file selection, the file data is read securely by the Main Process and transmitted back to the renderer via a `'file-opened'` message.
  - **Result:** The user can securely select a local file. The application's functionality is achieved without exposing the sensitive file system to the less secure renderer process.

5.3 Preload Script (`preload.js`)

This script acts as a secure bridge, running in a privileged context with access to both the renderer's `window` object and a subset of Node.js APIs.

- **API: `contextBridge.exposeInMainWorld('electronAPI', ...)`**
    - **Situation:** The sandboxed renderer process needs to communicate with the powerful main process. A direct channel is insecure.
    - **Task:** Create a safe, limited, and clearly defined API to expose specific backend functions to the renderer process.
    - **Action:** The `contextBridge` module is used to attach a custom `electronAPI` object to the renderer's global `window` object. This object contains only the functions we explicitly define, like `openFile()`. When called, these functions do not execute backend code directly; instead, they use the `ipcRenderer` module to send a message to the Main Process, requesting an action.
    - **Result:** The renderer can invoke backend functionality via a well-defined API (`window.electronAPI.openFile()`), maintaining a strong security boundary. The UI code remains isolated from powerful OS-level permissions.

5.4 Renderer Process (`renderer.js`)

This script governs the application's user interface, managing state, user input, and DOM manipulation.

- **State Management Variables (`pdfDoc`, `pageNum`, `scale`)**
    - **Situation:** The application must remember the user's current context, such as which document is open, the current page, and the zoom level.
    - **Task:** Establish variables to serve as a reliable "single source of truth" for the application's state.
    - **Action:** Simple JavaScript variables are declared at the top of the file to store the PDF.js document object (`pdfDoc`), the current page number (`pageNum`), and the zoom factor (`scale`). These variables are updated whenever the user interacts with the UI.
    - **Result:** The application maintains a consistent state. If we did not do this, for example, the application would forget the current page every time the user zoomed in or out, leading to a poor user experience.

- **Function: `renderPage(num)`**
    - **Situation:** A selected page from a loaded PDF document must be visually rendered for the user. A computer cannot "see" a PDF file directly; it must be drawn pixel by pixel.
    - **Task:** Retrieve a specified page from the `pdfDoc` object and render it onto the HTML `<canvas>` element, which is a dedicated drawing surface.

- **Action:** The function retrieves the page object using `pdfDoc.getPage(num)`. It then calculates the viewport dimensions based on the current `scale` and invokes the page's `.render()` method to draw it onto the canvas. A `pageRendering` flag is used as a gatekeeper to prevent the application from trying to draw two pages at once, which could cause visual glitches.
- **Result:** The selected PDF page is displayed accurately. The rendering process is managed to ensure a smooth and error-free user experience, even with rapid user input.

5.5 User Interface (`index.html` & `styles.css`)

- **`index.html`:** Defines the semantic structure (the "skeleton") of the user interface. It contains all UI controls (buttons, inputs) and, most importantly, the `<canvas>` element that serves as the rendering target for PDF pages. Each interactive element has a unique `id` so that `renderer.js` can reference it.
- **`styles.css`:** Provides the visual styling (the "skin") for the application. It defines the dark theme, the layout of components using modern CSS, and the appearance of all UI controls to ensure a clean and cohesive look.

6.0 FUTURE WORK

The following enhancements are planned for future development cycles to expand the application's capabilities and improve user experience:

- **Bookmark Functionality:** Implementation of a system to allow users to save references to specific pages within a document and navigate to them quickly.
- **Collapsible Library Panel:** An option to hide the file library sidebar, providing an immersive, distraction-free reading mode that maximizes the viewing area.
- **UI/UX Refinement:** A comprehensive redesign of the user interface, focusing on improving layout, iconography, accessibility, and visual feedback for user actions.
- **Annotation Tools:** Addition of tools for text highlighting, underlining, and adding personal text notes directly onto the PDF document.
- **Performance Optimization:** Further enhancements to the rendering engine and data handling to improve initial load times and reduce memory consumption, especially for large or complex documents.

7.0 LICENSE