Official Documentation: MediaSphere Docs Viewer

Version 1.0 | **Last Updated:** July 29, 2025

Abstract

The MediaSphere Docs Viewer is a desktop application providing a streamlined and unified interface for reading `.docx` and `.txt` files. As a component of the open-source MediaSphere Suite, it is engineered with Electron.js to deliver a cross-platform solution for document viewing. The application leverages powerful libraries to handle different file formats, including `mammoth.js` for high-fidelity `.docx` to HTML conversion. This document provides a comprehensive overview of the project's architecture, features, installation procedures, and component specifications.

1.0 INTRODUCTION

The MediaSphere Docs Viewer offers users a focused and feature-rich platform for reading common document formats. As part of the broader MediaSphere Suite, it adheres to the same principles of open-source development, performance, and user-centric design. The project is actively developed by a community of contributors, including members from GITAM (Deemed to be University).

The primary objective of the MediaSphere Suite is to create a single, unified application for all media formats. The Docs Viewer represents the project's solution for text-based documents.

2.0 KEY FEATURES

- **Multi-Format Support:** Natively opens and displays both Microsoft Word (`.docx`) and Plain Text (`.txt`) files.
- **High-Fidelity Rendering:** Converts `.docx` files to clean HTML, preserving formatting such as headings, lists, and tables.
- **Modern, Clean Interface:** A minimalist and dark-themed UI designed to minimize distractions and provide a comfortable reading environment.
- **Integrated File Management:** Users can open local document files directly through a native file dialog.

3.0 INSTALLATION AND EXECUTION

3.1 Prerequisites

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

3.2 Procedure

Execute the following commands in a terminal or command prompt:

Clone the source repository:
```
git clone https://github.com/AtheeqAhmedMJ/MediaSphereDocs.git
```

1.

Navigate to the project directory:
```
cd MediaSphereDocs
```

2.

Install dependencies:
```
npm install
```

3.

Execute the application:
```
npm start
```

4.

4.0 SYSTEM ARCHITECTURE

The application is built using the **Electron.js** framework, which enables the creation of desktop applications with web technologies. The architecture is bifurcated into two primary processes to ensure security and performance:

1. **Main Process (`main.js`):** The application's backend, running in a Node.js environment. It manages application windows and handles native operating system interactions.
2. **Renderer Process (`renderer.js`):** The application's frontend, responsible for rendering the user interface within a sandboxed Chromium window.

These processes communicate securely through a **Preload Script (`preload.js`)**, which selectively exposes backend functions to the frontend.

4.1 Data Flow Example: Opening a Document

To understand how the components work together, consider the step-by-step process when a user opens a `.docx` file:

1. **User Action:** The user clicks the "Open File" button in the Renderer Process (the user interface).

2. **Secure API Call:** The UI's JavaScript (`renderer.js`) calls the function `window.electronAPI.openFile()`. This function was securely exposed by the Preload Script.
3. **IPC Message:** The Preload Script sends a secure message (`'dialog:openFile'`) over the IPC channel to the Main Process.
4. **Native Action:** The Main Process (`main.js`) receives the message and executes its handler. This handler opens the operating system's native file selection dialog, filtered to show only `.docx` and `.txt` files.
5. **File Read:** After the user selects a file, the Main Process reads its contents into a raw data buffer.
6. **IPC Response:** The Main Process sends this data buffer back to the Renderer Process as the return value of the `handle` call.
7. **File Type Detection:** The Renderer Process receives the data. Its logic checks the file extension of the path it also received.
8. **Conversion & Rendering:** Since the file is a `.docx`, the data buffer is passed to the `mammoth.js` library. Mammoth converts the Word document into HTML.
9. **UI Update:** The resulting HTML is injected into the main `viewer <div>` on the page, displaying the formatted document to the user.

5.0 COMPONENT SPECIFICATION

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

5.1 Project Manifest (`package.json`)

This file serves as the project's configuration manifest, defining its metadata and dependencies.

- **`"main": "main.js"`:** Specifies the entry point for the Electron application.
- **`"scripts": { "start": "electron ." }`:** Defines the `npm start` command for easy execution.
- **`"dependencies": { "mammoth": "^1.7.1" }`:** Declares **Mammoth.js** as a critical dependency for converting `.docx` files.

5.2 Main Process (`main.js`)

This script controls the application's lifecycle and backend operations.

- **Function: `createWindow()`**
  - **Purpose:** To initialize and configure the main application window that the user interacts with.
  - **Action:** A `BrowserWindow` instance is created. The `webPreferences.preload` option is set to load `preload.js`, establishing a secure communication bridge to the renderer process.

    - **Result:** A native desktop window is created, ready to load the `index.html` user interface.
- **IPC Handler: `ipcMain.handle('dialog:openFile', ...)`**
    - **Purpose:** To provide a secure way for the UI to request access to the local file system.
    - **Action:** An Inter-Process Communication handler is established using `ipcMain.handle()`. When the UI requests to open a file, this function executes, showing a native file dialog filtered for `.docx` and `.txt` files. It returns the path and raw data buffer of the selected file to the renderer process.
    - **Result:** The user can securely select a local document file without exposing the entire file system to the sandboxed UI.

5.3 Preload Script (`preload.js`)

This script acts as a secure bridge between the frontend and backend.

- **API: `contextBridge.exposeInMainWorld('electronAPI', ...)`**
    - **Purpose:** To securely expose specific backend functions to the renderer process.
    - **Action:** The `contextBridge` module attaches a custom `electronAPI` object to the UI's global `window` object. This object contains the `openFile()` function, which internally invokes the `dialog:openFile` IPC handler in the main process.
    - **Result:** The UI can call `window.electronAPI.openFile()` to trigger a file dialog, maintaining a strong security boundary between the two processes.

5.4 Renderer Process (`renderer.js`)

This script governs the application's user interface, managing the document rendering and user interactions.

- **File Handling Logic**
    - **Purpose:** To provide a flexible way to handle different document formats within the same application.
    - **Action:** When the `openFile()` function is called and returns data, a `switch` statement is used to inspect the file extension (`path.extname`). Based on the extension (`.docx` or `.txt`), the code executes a different block of logic specifically designed for that file type.
    - **Result:** The application is modular and extensible. To add support for a new file type in the future (e.g., `.md`), a developer would only need to add a new `case` to the switch statement and a corresponding rendering function.
- **`.docx` Rendering with Mammoth.js**

    - **Purpose:** To convert the complex, proprietary `.docx` format into standard HTML that a web browser can display. A `.docx` file is essentially a zip archive of XML files, which cannot be rendered directly.
    - **Action:** The raw data `buffer` of the `.docx` file is passed to the `mammoth.convertToHtml({ buffer: data })` function. Mammoth.js unzips the file in memory, parses the XML to understand the document's structure (headings, paragraphs, lists), and converts it into semantic HTML.
    - **Result:** A clean HTML representation of the Word document is generated. This HTML is then set as the `innerHTML` of the main `viewer <div>`, preserving essential formatting and making the document readable.
- **`.txt` Rendering**
    - **Purpose:** To display the contents of a simple plain text file while preserving its original formatting, like line breaks and spacing.
    - **Action:** For `.txt` files, the raw data `buffer` is first converted into a JavaScript string. By default, HTML collapses whitespace. To prevent this and show the text exactly as it is in the file, the entire string is wrapped within `<pre>` (preformatted text) tags before being inserted into the `viewer <div>`.
    - **Result:** Plain text files are displayed with their original whitespace and line breaks intact, ensuring code snippets, poetry, or other formatted text is rendered correctly.

6.0 LICENSE

This project is distributed under the terms of the **MIT License**. The full license text is available in the project repository.