Implementation Report: Module 1 - Input Handling

Project: Inclusive Reading Aid for Dyslexic & Visually Implemented Users **Module:** Point 1 - Input Handling **Status:** Completed **Date:** 2025-11-08

1.0 Introduction

This report details the technical methodology for constructing the **Input Handling** module. This module serves as the primary data-ingestion layer for the application, providing multiple pathways for users to import text from various sources.

The objective is to allow users to load content from text files, PDF documents, and image-based sources (via Optical Character Recognition), as specified in the project proposal. This module bridges the gap between external content and the application's frontend display (Module 2).

2.0 Core Technologies & Dependencies

This module introduces several critical backend dependencies:

1. **PyQt6:**
   - `QMenuBar`, `QMenu`, `QAction`: Used to create the main application menu bar for file and tool selection.
   - `QFileDialog`: Used to create native OS file-opening dialogs for file selection.
2. **PyPDF2:** The designated library for reading and extracting text content from `.pdf` files.
3. **Pillow (PIL):** The Python Imaging Library, used for:
   - Opening and processing image files (`.png`, `.jpg`, etc.) via `Image.open()`.
   - Capturing a fullscreen screenshot via `ImageGrab.grab()`.
4. **pytesseract:** The Python wrapper for Google's Tesseract-OCR engine. It is used to perform all OCR tasks.
5. **Tesseract-OCR Engine (External Dependency):** This module is **critically dependent** on the Tesseract-OCR engine being installed and accessible on the host operating system's PATH. The `pytesseract` library will fail without it.

3.0 Implementation Methodology

The module's functionality was integrated by creating a main menu bar and connecting its actions to new backend processing functions.

3.1 GUI Construction (Menu Bar)

A `_create_menu_bar` helper function was added to the `DyslexiaReaderApp` class to construct the main menu.

- A `QMenuBar` is created and attached to the `QMainWindow`.
- A **"File" menu** is created, containing `QAction`s for "Open Text File...", "Open PDF File...", "Open Image File (OCR)...", and "Exit".
- A **"Tools" menu** is created, containing a `QAction` for "Capture Fullscreen (OCR)".

- Each `QAction` 's `triggered` signal is connected to a corresponding handler function (e.g., `self.open_text_file`).

3.2 File Input Logic

Three distinct functions were created to handle file-based input:

1. `open_text_file()` **(for** `.txt` **):**
   - Uses `QFileDialog.getOpenFileName()` to prompt the user for a `.txt` file.
   - If a file is selected, it is opened using a standard Python `open()` context manager with `encoding='utf-8'` .
   - The file's content is read and placed into the main text area via `self.text_area.setPlainText()` .
2. `open_pdf_file()` **(for** `.pdf` **):**
   - Uses `QFileDialog.getOpenFileName()` to prompt the user for a `.pdf` file.
   - A `PyPDF2.PdfReader` object is instantiated with the selected file path.
   - The function iterates through all `reader.pages` , calls `page.extract_text()` on each, and concatenates the results into a single string.
   - The final extracted text is set in `self.text_area` .
3. `open_image_file()` **(for** `.png` **,** `.jpg` **, etc.):**
   - Uses `QFileDialog.getOpenFileName()` to prompt the user for an image file.
   - The selected file path is opened using `PIL.Image.open()` .
   - The resulting image object is passed directly to `pytesseract.image_to_string()` .
   - The text returned by Tesseract is set in `self.text_area` .

3.3 Screen Capture (OCR) Logic

A more complex OCR function, `capture_fullscreen_ocr()` , was implemented:

1. **Window Hiding:** The main application window is hidden via `self.hide()` to prevent it from appearing in its own screenshot.
2. **Screen Grab:** `PIL.ImageGrab.grab()` is called to capture the entire screen and store it as an image object.
3. **Window Restore:** The main window is immediately restored using `self.show()` .
4. **OCR Processing:** The captured screenshot (image object) is passed to `pytesseract.image_to_string()` .
5. **Display:** The extracted text is placed in `self.text_area` .

3.4 Error Handling

Robust `try...except` blocks are wrapped around all input/OCR logic.

- A specific `except pytesseract.TesseractNotFoundError` block is implemented for all OCR functions. If Tesseract is not installed, this error is caught, and a user-friendly message is displayed in the `self.text_area` directing the user to install the engine.

- A general `except Exception as e` block catches all other file-reading or processing errors (e.g., corrupted files, permissions issues) and reports them to the user.

4.0 Summary of Features Implemented

- [x] **File Menu:** A full "File" menu is present for loading content.
- [x] **Text File Input:** Users can successfully open and read `.txt` files.
- [x] **PDF File Input:** Users can successfully open and extract text from `.pdf` files.
- [x] **Image File OCR:** Users can open image files, and text is extracted via OCR.
- [x] **Screen Capture OCR:** A "Tools" menu provides a function to capture the screen and extract text via OCR.
- [x] **Error Handling:** The application correctly handles and reports errors, especially the critical "Tesseract not found" error.

5.0 Conclusion

Module 1 is fully implemented and functional. The application is no longer reliant on placeholder text and can now be populated with user-provided content from a wide variety of specified sources. The successful completion of this module provides the necessary text for all other processing and output modules.

The next logical step is the implementation of **Module 3 (Audio Support)**, which will process the text now being loaded into the application.