requirements

Requirements Document

Introduction

Kiro2PDF is a command-line utility designed to convert Kiro specification markdown files into a consolidated, printable PDF document. The tool aims to provide a simple way for users to generate professional-looking documentation from their Kiro markdown specs, with proper formatting, navigation, and organization.

Requirements

Requirement 1: File Conversion

User Story: As a Kiro user, I want to convert markdown specification files into PDF format, so that I can share and print my specifications in a professional format.

Acceptance Criteria

1. WHEN the user provides one or more markdown files THEN the system SHALL convert them into a single PDF document
2. WHEN conversion is complete THEN the system SHALL save the PDF to the specified output location
3. IF a markdown file cannot be read THEN the system SHALL display an error message and continue processing other files
4. WHEN processing multiple files THEN the system SHALL maintain the order of files as specified by the user

Requirement 2: Document Organization

User Story: As a Kiro user, I want the generated PDF to be well-organized with a table of contents, so that I can easily navigate through the document.

Acceptance Criteria

1. WHEN generating the PDF THEN the system SHALL create a table of contents (TOC)
2. WHEN creating the TOC THEN the system SHALL include an entry for each input file
3. WHEN creating the TOC THEN the system SHALL use the filename (without extension) as the section title
4. WHEN processing markdown content THEN the system SHALL respect the heading hierarchy in the original markdown
5. WHEN the user specifies a TOC level THEN the system SHALL only include headings up to that level in the TOC

Requirement 3: Customization Options

User Story: As a Kiro user, I want to customize the PDF output, so that I can control the appearance and size of the generated document.

Acceptance Criteria

1. WHEN the user provides a CSS file THEN the system SHALL apply the custom styling to the PDF
2. IF the CSS file cannot be read THEN the system SHALL display an error message and proceed without custom styling
3. WHEN the user requests optimization THEN the system SHALL reduce the PDF file size
4. WHEN generating the PDF THEN the system SHALL set appropriate metadata (title, author)

Requirement 4: Command-Line Interface

User Story: As a Kiro user, I want a simple command-line interface, so that I can easily use the tool in my workflow.

Acceptance Criteria

1. WHEN the tool is executed THEN the system SHALL accept command-line arguments for input files and output file
2. WHEN the tool is executed THEN the system SHALL provide optional arguments for customization (TOC level, optimization, CSS)
3. WHEN processing files THEN the system SHALL display progress information
4. WHEN an error occurs THEN the system SHALL display informative error messages
5. WHEN the conversion is successful THEN the system SHALL display a success message with the output file path

design

Design Document

Overview

Kiro2PDF is a command-line utility that converts Kiro specification markdown files into a consolidated PDF document. The tool is designed to be simple to use while providing customization options for the output. It leverages the `markdown-pdf` library to handle the conversion process and adds features like automatic table of contents generation and custom styling.

Architecture

The architecture of Kiro2PDF is straightforward, following a command-line application pattern:

1. **Command-Line Interface**: Handles user input through command-line arguments
2. **PDF Generation Core**: Processes markdown files and generates the PDF
3. **External Library Integration**: Leverages the `markdown-pdf` library for the actual conversion

```
graph TD
    A[Command-Line Interface] --> B[PDF Generation Core]
    B --> C[markdown-pdf Library]
    B --> D[File System Operations]
```

Components and Interfaces

Command-Line Interface

The command-line interface is implemented using Python's `argparse` library. It defines the following parameters:

- `input_files`: One or more paths to input Markdown files (required, positional arguments)
- `-o, --output`: The path for the output PDF file (required)
- `-t, --toc-level`: Maximum heading level to include in the Table of Contents (optional, default=6)
- `-z, --optimize`: Flag to optimize the PDF file size (optional)
- `-c, --css`: Path to a CSS file for custom styling (optional)

PDF Generation Core

The core functionality is implemented in the `create_pdf_from_markdown_files` function, which:

1. Initializes a `MarkdownPdf` object with the specified TOC level and optimization

settings
2. Processes each input markdown file:
    - Reads the file content
    - Extracts the filename to use as a section title
    - Prepends a top-level heading to ensure proper TOC entry
    - Adds the content as a section to the PDF
3. Sets PDF metadata
4. Saves the PDF to the specified output location

External Library Integration

The tool relies on the `markdown-pdf` library, specifically using:

- `MarkdownPdf`: Main class for PDF generation
- `Section`: Class representing a section in the PDF

Data Models

Input Data

- **Markdown Files**: Text files with markdown formatting
- **CSS File** (optional): Text file with CSS styling rules

Output Data

- **PDF Document**: A structured PDF with:
    - Table of Contents
    - Sections for each input file
    - Metadata (title, author)

Error Handling

The application implements the following error handling strategies:

1. **File Not Found**: If an input markdown file or CSS file is not found, the application displays a warning message and continues processing other files
2. **File Reading Errors**: If there's an error reading a file, the application displays an error message and continues processing other files
3. **PDF Generation Errors**: If there's an error during PDF generation, the application displays an error message

All errors are logged to the console with descriptive messages to help users troubleshoot issues.

Testing Strategy

Unit Testing

Unit tests should be implemented for:

1. **Command-Line Argument Parsing**: Verify that arguments are correctly parsed and validated
2. **File Processing**: Test the handling of various file scenarios (valid files, missing files, files with errors)
3. **Error Handling**: Verify that errors are properly caught and reported

Integration Testing

Integration tests should verify:

1. **End-to-End Functionality**: Test the complete flow from command-line input to PDF generation
2. **Library Integration**: Verify correct interaction with the `markdown-pdf` library

Manual Testing

Manual testing should focus on:

1. **Output Quality**: Verify that the generated PDFs have correct formatting, TOC, and styling
2. **Edge Cases**: Test with very large files, files with complex markdown, and various customization options

tasks

Implementation Plan

- [ ] 1. Set up project structure and dependencies
    - Create the basic project structure with main script file
    - Define dependencies in requirements.txt
    - *Requirements: 1.1, 4.1*
- [ ] 2. Implement command-line interface
    - [ ] 2.1 Create argument parser with required arguments
        - Implement input file arguments (multiple files support)
        - Implement output file argument
        - Write unit tests for basic argument parsing
        - *Requirements: 4.1, 4.2*
    - [ ] 2.2 Add optional customization arguments
        - Implement TOC level argument
        - Implement optimization flag
        - Implement CSS file argument
        - Write unit tests for optional arguments
        - *Requirements: 3.1, 3.3, 4.2*
- [ ] 3. Implement core PDF generation functionality
    - [ ] 3.1 Create function to process markdown files
        - Implement file reading with error handling
        - Extract filenames for section titles
        - Write unit tests for file processing
        - *Requirements: 1.1, 1.3, 2.3*
    - [ ] 3.2 Implement PDF generation with markdown-pdf
        - Initialize MarkdownPdf with user settings
        - Add sections for each markdown file
        - Set PDF metadata
        - Write unit tests for PDF generation
        - *Requirements: 1.1, 1.2, 2.1, 2.4, 3.4*
- [ ] 4. Implement table of contents generation
    - Create logic to generate TOC entries for each file
    - Implement TOC level filtering
    - Write unit tests for TOC generation
    - *Requirements: 2.1, 2.2, 2.5*
- [ ] 5. Implement custom styling support

    - Add CSS file loading with error handling
    - Apply custom styling to PDF sections
    - Write unit tests for CSS handling
    - *Requirements: 3.1, 3.2*

- [ ] 6. Implement progress reporting and error handling

    - Add progress messages during processing
    - Implement comprehensive error handling
    - Add success message with output file path
    - Write unit tests for error scenarios
    - *Requirements: 1.3, 3.2, 4.3, 4.4, 4.5*

- [ ] 7. Implement PDF optimization

    - Add optimization option to reduce file size
    - Write unit tests for optimization
    - *Requirements: 3.3*

- [ ] 8. Create comprehensive README documentation

    - Document installation instructions
    - Document usage examples
    - Document all available options
    - *Requirements: 4.1, 4.2*

- [ ] 9. Final integration testing

    - Test end-to-end functionality with various inputs
    - Test with edge cases (large files, complex markdown)
    - Fix any identified issues
    - *Requirements: 1.1, 1.2, 1.3, 1.4*