System Design Document: Simple Document Processor

Your Name

September 17, 2025

Contents

1 Introduction

This document presents a comprehensive system design for the **Simple Document Processor**, a modular backend service for extracting and processing text from TXT, PDF, and DOCX files. The system is designed for extensibility, robustness, and ease of integration into larger pipelines such as RAG (Retrieval-Augmented Generation) or document analytics.

2 Requirements

2.1 Functional Requirements

- Support extraction of text from TXT, PDF, and DOCX files.
- Clean and normalize extracted text.
- Split text into overlapping chunks for downstream processing.
- Extract keywords from the text.
- Provide clear error messages for unsupported formats or processing failures.

2.2 Non-Functional Requirements

- Minimal dependencies and lightweight operation.
- Easy extensibility for new file types.
- Robust error handling and logging.
- Stateless and secure processing.
- Suitable for CLI, API, or library integration.

3 Architecture

3.1 Overview

The system follows a modular, layered architecture:

- **API Layer**: Handles user requests (REST API, CLI, or direct function calls).
- **Processing Layer**: Contains the `SimpleDocumentProcessor` and file-type-specific processors.
- **Data Layer**: Manages file I/O and extracted data.

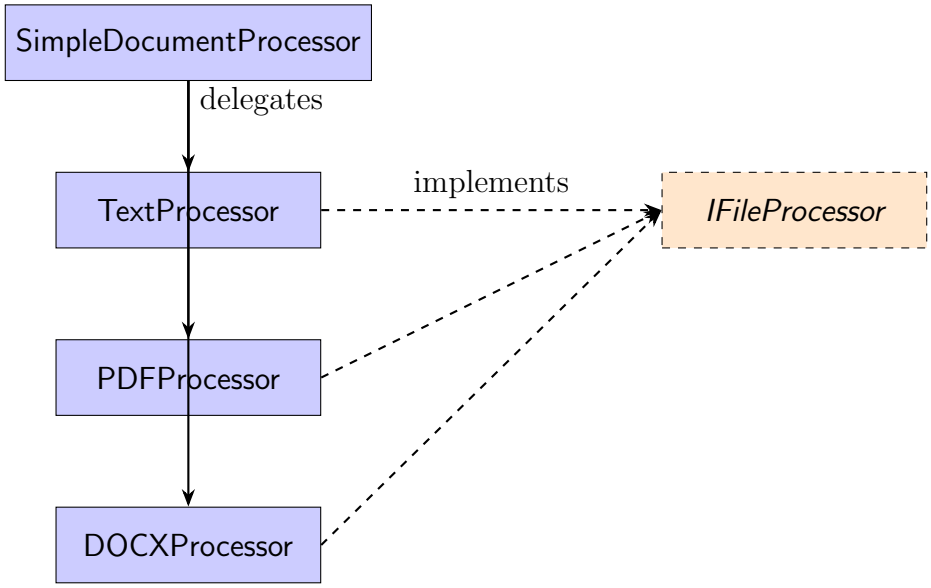
3.2 Workflow

1. User submits a document.
2. System detects file type and delegates to the appropriate processor.
3. Extracted text is cleaned and normalized.
4. Cleaned text is chunked.
5. Keywords are extracted.
6. Results are returned to the user or calling system.

3.3 Extensibility and Security

- New file types can be added by registering new processors.
- Files are processed in memory or isolated temp directories.
- No persistent storage unless explicitly configured.
- Input validation and error handling at each layer.

3.4 Static UML Diagram (Component/Class)



4 Data Design

4.1 Input Data

- **file_path**: string (absolute or relative)
- **file_type**: string (`.txt`, `.pdf`, `.docx`)

4.2 Internal Data Structures

- **Raw Text**: Unprocessed text extracted from the document.
- **Cleaned Text**: Normalized text.
- **Chunks**: List of text segments (overlapping, configurable size).
- **Keywords**: List of extracted keywords.

4.3 Output Data

- `text`: Cleaned text.
- `original_length`: Length of raw text.
- `cleaned_length`: Length of cleaned text.
- `file_type`: File extension.
- `word_count`: Number of words.
- `chunks`: List of text chunks.
- `keywords`: List of keywords.

4.4 Data Flow

1. Input: File path and extension.
2. Processing: Extraction → Cleaning → Chunking → Keyword Extraction.
3. Output: Structured dictionary.

5 Component Breakdown

5.1 SimpleDocumentProcessor

- Orchestrates file processing, cleaning, chunking, and keyword extraction.
- Maintains registry of supported file processors.
- Handles error logging and reporting.

5.2 File Type Processors

- **TextProcessor**: Reads and decodes TXT files.
- **PDFProcessor**: Extracts text from PDFs using `pdfplumber`.
- **DOCXProcessor**: Extracts text from DOCX files using `python-docx`.

5.3 Text Cleaner

- Removes excessive whitespace and normalizes newlines.
- Strips leading/trailing whitespace.

5.4 Chunker

- Splits cleaned text into overlapping chunks.
- Uses sentence boundaries for natural splits.
- Handles edge cases (e.g., very long sentences).

5.5 Keyword Extractor

- Uses regex to extract candidate keywords.
- Filters out common stopwords.
- Ranks keywords by frequency.

5.6 Logging and Error Handling

- All exceptions are logged with context.
- User-facing errors are descriptive and actionable.

6 Technologies and Rationale

6.1 Programming Language

- **Python 3.x**: Readability, ecosystem, and strong support for text/document processing.

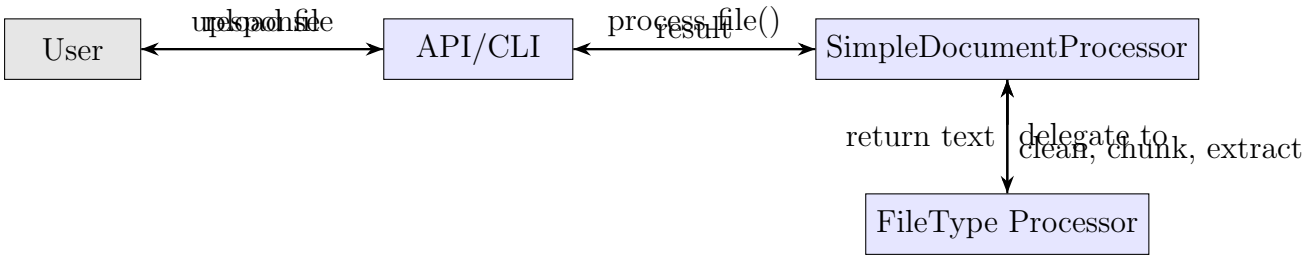
6.2 Libraries

- **pdfplumber**: Lightweight PDF text extraction.
- **python-docx**: DOCX parsing.
- **re**: Standard regex for text cleaning and keyword extraction.
- **logging**: Robust error/event logging.

6.3 Why These Choices?

- Minimal dependencies, lightweight, and cross-platform.
- Easy extensibility for new file types.
- No database required; operates on files directly.
- Suitable for CLI, API, or library integration.

7 Dynamic Workflow (Sequence UML)



8 Design Rationale

8.1 Simplicity and Maintainability

- Clear separation of concerns.
- Each file type processor is isolated for easy maintenance and extension.
- Minimal dependencies reduce compatibility risks.

8.2 Performance

- Processes files in memory for speed (suitable for small/medium documents).
- Chunking enables parallel or incremental downstream processing.

8.3 Extensibility

- New file types can be supported by implementing and registering new processors.
- Additional processing steps (e.g., language detection, summarization) can be added.

8.4 Error Handling and Security

- Comprehensive logging for all processing steps.
- Graceful handling of encoding issues and corrupt files.
- No persistent storage of sensitive data unless configured.
- Input validation and isolation of file processing.

8.5 Use Cases

- Preprocessing for RAG pipelines.
- Document ingestion for search engines or knowledge bases.
- Standalone document analysis or keyword extraction.

9 Conclusion

This design provides a robust, extensible, and lightweight document processing system suitable for basic file types and easy integration into larger data pipelines or applications.