Low-Level Design (LLD) for "Document Processing System"

Page 1

Table of Contents

Page 2

1. Introduction

1.1 Purpose: This Low-Level Design (LLD) document details the implementation specifics of the Document Processing System (DPS). It expands upon the High-Level Design document, providing the necessary information for developers to implement the system.

1.2 Scope: This document covers the detailed design of all DPS modules, including data structures, algorithms, database schema, user interface elements, security measures, error handling, and testing strategies.

1.3 Audience: This document is intended for software developers, database administrators, and testers involved in the DPS project.

1.4 References:

* High-Level Design Document for Document Processing System

* [List any relevant external libraries or APIs]

1.5 Definitions:

* **DPS:** Document Processing System

* **API:** Application Programming Interface

* **OCR:** Optical Character Recognition

Page 3

2. System Overview

The Document Processing System (DPS) automates the processing of incoming documents, extracting key information, and routing them to the appropriate departments. The system handles various document types (PDF, DOCX, TXT, etc.) and formats, ensuring efficient and accurate processing. It integrates with existing CRM and ERP systems to streamline workflows and improve data management. The system will be a three-tiered architecture (Presentation, Application, Data) deployed on a cloud platform (e.g., AWS, Azure).

Page 4

3. Detailed Design

3.1 Module Descriptions:

* **Document Upload Module:** Handles document upload via a web interface. Validates file types and sizes. Uses a temporary storage location (e.g., AWS S3) for uploaded files before processing. Implemented using a RESTful API endpoint.

* **Document Processing Module:** Performs OCR (using Tesseract OCR library) on image-based documents. Extracts key data using regular expressions and potentially machine learning models (if specified in HLD). Converts documents to a standardized format (e.g., plain text). Uses a message

queue (e.g., RabbitMQ, Kafka) for asynchronous processing.

* **Data Extraction Module:** This module is a sub-module of the Document Processing Module. It uses a rule-based engine (defined by configuration files) and regular expressions to extract specific data points from processed documents. It can be extended to incorporate machine learning models for more advanced extraction.

* **Data Storage Module:** Stores processed documents, extracted data, and metadata in a relational database (PostgreSQL). Uses a database connection pool for efficient resource management.

* **Reporting Module:** Generates reports in various formats (PDF, CSV, etc.) using a reporting library (e.g., JasperReports). Provides functionality for users to customize reports based on specific data fields.

* **User Management Module:** Manages user accounts, roles, and permissions using a secure authentication and authorization mechanism (e.g., OAuth 2.0 with JWT). Integrates with an existing identity provider (if applicable).

* **Workflow Management Module:** (Optional, depending on HLD) Manages the routing of documents to different departments based on extracted data. Uses a workflow engine (e.g., Camunda) to define and execute workflows.

Page 5

3.1 Module Descriptions (continued):

* **API Gateway Module:** Acts as a central point of entry for all API requests. Handles request routing, authentication, and authorization. Uses API Gateway services offered by cloud provider (AWS API Gateway, Azure API Management).

* **Logging and Monitoring Module:** Collects logs from all modules and sends them to a centralized logging system (e.g., ELK stack, CloudWatch). Provides monitoring dashboards for system performance and error tracking. Uses application performance monitoring (APM) tools for real-time monitoring.

* **Configuration Management Module:** Manages system configurations (database connections, API keys, etc.) through a central configuration repository (e.g., Consul, etcd). Allows for easy modification of configurations without requiring code changes.

Page 6

3.1 Module Descriptions (continued):

* **External System Integration Module:** Handles communication with external systems (CRM, ERP) via their respective APIs. Uses appropriate integration patterns (e.g., REST, SOAP). Implements error handling and retry mechanisms for reliable communication.

Page 7

3.2 Class Diagrams:

(a) Document Processing Module

~~~
```plantuml
@startuml
class Document {
    - documentId : int
    - fileName : String
    - filePath : String
    - fileType : String
    + getDocumentId() : int
    + getFileName() : String
    + getFileType() : String
    + getProcessedData() : Map<String, String>
}

class OCRProcessor {
    + processOCR(Document) : String
}

class DataExtractor {
    - extractionRules : List<Rule>
    + extractData(String) : Map<String, String>
}

class Rule {
~~~

```
    - pattern : String
    - fieldName : String
    + matches(String) : boolean
    + extractValue(String) : String
}

Document "1" -- "1" OCRProcessor : uses
OCRProcessor "1" -- "1" DataExtractor : uses
DataExtractor "1" -- "*" Rule : uses

@enduml
```

(b) Data Storage Module

```plantuml
@startuml
class DocumentData {
    - documentId : int
    - extractedData : Map<String, String>
    + getDocumentId() : int
    + getExtractedData() : Map<String, String>
}

class DocumentMetadata {
    - documentId : int
    - uploadDate : Date
    - processedDate : Date

    + getDocumentId() : int

    + getUploadDate() : Date

    + getProcessedDate() : Date

}


class Database {

    + storeDocumentData(DocumentData) : void

    + storeDocumentMetadata(DocumentMetadata) : void

    + retrieveDocumentData(int) : DocumentData

    + retrieveDocumentMetadata(int) : DocumentMetadata

}


DocumentData "1" -- "1" Database : stored in

DocumentMetadata "1" -- "1" Database : stored in


@enduml
```

Page 8

(c) User Management Module

```plantuml
@startuml
```

```
class User {
    - userId : int
    - username : String
    - password : String
    - roles : List<String>
    + getUserId() : int
    + getUsername() : String
    + getRoles() : List<String>
}

class AuthenticationService {
    + authenticate(String, String) : boolean
    + authorize(User, String) : boolean
}

class AuthorizationService {
    + isAuthorized(User, String) : boolean
}

User "1" -- "1" AuthenticationService : uses
User "1" -- "1" AuthorizationService : uses

@enduml
```

Page 9

(d) Simplified Overview Class Diagram

```plantuml
@startuml
class UserInterface
class DocumentUploadModule
class DocumentProcessingModule
class DataStorageModule
class ReportingModule
class UserManagementModule

UserInterface -- DocumentUploadModule
DocumentUploadModule -- DocumentProcessingModule
DocumentProcessingModule -- DataStorageModule
DocumentProcessingModule -- ReportingModule
UserInterface -- UserManagementModule

@enduml
```

Page 10

3.3 Sequence Diagrams:

(a) Document Upload and Processing:

~~~
```plantuml
@startuml
actor User
participant UserInterface
participant DocumentUploadModule
participant DocumentProcessingModule
participant DataStorageModule

User -> UserInterface : Uploads Document
activate UserInterface
UserInterface -> DocumentUploadModule : Send Document
activate DocumentUploadModule
DocumentUploadModule -> DataStorageModule : Store temporary file
activate DataStorageModule
DataStorageModule --> DocumentUploadModule : Success/Failure
deactivate DataStorageModule
DocumentUploadModule --> UserInterface : Upload status
deactivate DocumentUploadModule
UserInterface --> User : Display status
deactivate UserInterface

UserInterface -> DocumentProcessingModule : Process Document
activate DocumentProcessingModule
DocumentProcessingModule -> DataStorageModule : Store processed data & document
activate DataStorageModule
DataStorageModule --> DocumentProcessingModule : Success/Failure
~~~

deactivate DataStorageModule

DocumentProcessingModule --> UserInterface : Processing Complete

deactivate DocumentProcessingModule

UserInterface --> User : Display results

@enduml

~~~

Page 11

(b) Report Generation:

```plantuml
@startuml
actor User
participant UserInterface
participant ReportingModule
participant DataStorageModule

User -> UserInterface : Requests Report
activate UserInterface
UserInterface -> ReportingModule : Generate Report request
activate ReportingModule
ReportingModule -> DataStorageModule : Retrieve Data
activate DataStorageModule
DataStorageModule --> ReportingModule : Report Data
deactivate DataStorageModule
~~~

ReportingModule --> UserInterface : Report Generated

deactivate ReportingModule

UserInterface --> User : Display Report

deactivate UserInterface

@enduml

~~~

**Page 12**

**3.4 State Diagrams:**

**(a) Document Processing State Diagram:**

```plantuml
@startuml
state "Uploaded" as Uploaded
state "Processing" as Processing
state "Processed" as Processed
state "Error" as Error

[*] --> Uploaded
Uploaded --> Processing : Processing started
Processing --> Processed : Processing complete
Processing --> Error : Processing error
Processed --> [*]
Error --> [*]
~~~

@enduml

**Page 13**

**(b) User State Diagram (Simplified):**

```plantuml
@startuml
state "Logged Out" as LoggedOut
state "Logged In" as LoggedIn

[*] --> LoggedOut
LoggedOut --> LoggedIn : Login Successful
LoggedIn --> LoggedOut : Logout
LoggedIn --> LoggedIn : Upload Document
LoggedIn --> LoggedIn : View Reports

@enduml
```

Page 14

3.5 Activity Diagrams:

(a) Document Processing Activity Diagram:

```plantuml
@startuml
start
:Receive Document;
:Validate Document;
if (Valid?) then (yes)
    :Perform OCR (if needed);
    :Extract Data;
    :Store Data;
    :Generate Report (if needed);
else (no)
    :Reject Document;
endif
:Return Result;
stop
@enduml
```

Page 15

(b) User Login Activity Diagram:

```plantuml
@startuml
start
:User enters credentials;
:Authenticate user;
if (Authentication successful?) then (yes)
    :Grant access;
    :Redirect to dashboard;
else (no)
    :Display error message;
endif
stop
@enduml
```

Page 16

4. Data Design

4.1 Data Structures:

* **Document:** `{documentId (INT, primary key), fileName (VARCHAR), filePath (VARCHAR), fileType (VARCHAR), uploadDate (TIMESTAMP), processedDate (TIMESTAMP), status (VARCHAR)}`

* **ExtractedData:** `{documentId (INT, foreign key), fieldName (VARCHAR), fieldValue (TEXT)}`

* **User:** `{userId (INT, primary key), username (VARCHAR), password (VARCHAR), role (VARCHAR)}`

* **Report:** `{reportId (INT, primary key), documentId (INT, foreign key), reportType (VARCHAR), generationDate (TIMESTAMP), reportData (BLOB)}`

Page 17

4.2 Database Design:

The system will utilize a PostgreSQL database. The database schema will consist of the following tables (see Appendix for a more detailed schema with constraints and indexes):

* **documents:** Stores metadata about uploaded documents.
* **extracted_data:** Stores the key-value pairs extracted from documents.
* **users:** Stores information about system users.
* **reports:** Stores generated reports.
* **logs:** Stores system logs for debugging and monitoring.

Page 18

(Relational Database Schema - Simplified)

```
```

documents:

document_id (INT, PRIMARY KEY)

file_name (VARCHAR)

file_path (VARCHAR)

upload_date (TIMESTAMP)

status (VARCHAR)

extracted_data:

id (INT, PRIMARY KEY)

document_id (INT, FOREIGN KEY referencing documents)

field_name (VARCHAR)

field_value (TEXT)

users:

user_id (INT, PRIMARY KEY)

username (VARCHAR)

password (VARCHAR)

role (VARCHAR)

reports:

report_id (INT, PRIMARY KEY)

document_id (INT, FOREIGN KEY referencing documents)

report_type (VARCHAR)

generation_date (TIMESTAMP)

report_data (BLOB)

logs:

    log_id (INT, PRIMARY KEY)
    timestamp (TIMESTAMP)
    level (VARCHAR)
    message (TEXT)

Page 19

4.3 Data Flow Diagram:

```mermaid
graph LR
    A[User] --> B(Document Upload Module);
    B --> C(Document Processing Module);
    C --> D(Data Extraction Module);
    D --> E(Data Storage Module);
    C --> F(Reporting Module);
    F --> G[User/External System];
    E --> F;
    B --> H(User Management Module);
    H --> B;
    C --> I(Logging Module);
    I --> J[Monitoring Dashboard];
```

Page 20

5. Interface Design

5.1 User Interface:

The user interface will be a web application built using React.js and Material-UI. Key features include:

* **Document Upload:** A drag-and-drop interface for uploading documents. Displays upload progress and status.
* **Report Generation:** Allows users to generate reports based on selected criteria. Supports various report formats (PDF, CSV).
* **Data Visualization:** Provides charts and graphs to visualize extracted data.
* **User Management:** Allows administrators to manage user accounts and permissions.
* **Monitoring Dashboard:** Displays system performance metrics and error logs.

The UI will be responsive and accessible across different devices.

Page 21

5.2 External Interfaces:

* **CRM Integration:** The system will integrate with the company's CRM system via a REST API to update customer records with extracted data. The integration will utilize OAuth 2.0 for authentication and authorization.

* **ERP Integration:** Similar to the CRM integration, a REST API will be used to update relevant data in the ERP system. Error handling and retry mechanisms will be implemented to ensure data consistency.

* **Third-party OCR API:** (If not using an embedded OCR library) The system will utilize a third-party OCR API (e.g., Google Cloud Vision API) for optical character recognition if needed. Appropriate API keys and security measures will be implemented.

Page 22

6. Algorithms and Complexity Analysis:

* **OCR Processing:** The complexity of OCR processing depends on the size and complexity of the document. Using Tesseract, the time complexity is approximately O(n), where n is the number of pixels in the image. Optimization techniques like image pre-processing can improve performance.

* **Data Extraction:** The complexity of data extraction depends on the complexity of regular expressions and the number of rules. In the worst case, it could be O(n*m), where n is the length of the text and m is the number of rules. Efficient regular expression engines can minimize this complexity.

* **Database Operations:** Database operations like insertion and retrieval have complexities that vary based on the database system and indexing. Generally, they are O(log n) for indexed searches and O(n) for linear scans.

Page 23

6. Algorithms and Complexity Analysis (continued):

* **Report Generation:** The complexity of report generation depends on the size of the data and the complexity of the report template. Using a reporting library like JasperReports, this complexity can be optimized to avoid O(n^2) scenarios.

* **Authentication and Authorization:** The complexity of authentication and authorization depends on the chosen method. Using JWT, the complexity is relatively low, primarily involving token verification (O(1)).

Page 24

7. Security Design:

* **Authentication:** OAuth 2.0 with JWT will be used for secure user authentication.
* **Authorization:** Role-based access control (RBAC) will be implemented to restrict access to sensitive data and functionalities based on user roles.
* **Data Encryption:** Data at rest will be encrypted using AES-256. Data in transit will be secured using HTTPS.
* **Input Validation:** All user inputs will be validated to prevent SQL injection and cross-site scripting (XSS) attacks.
* **Regular Security Audits:** Regular security audits and penetration testing will be conducted to

identify and address potential vulnerabilities.

Page 25

8. Error Handling and Logging:

* **Exception Handling:** Try-catch blocks will be used to handle exceptions gracefully. Error messages will be logged and reported to administrators.
* **Logging:** A centralized logging system (e.g., ELK stack) will be used to record all system events, including errors, warnings, and informational messages.
* **Error Reporting:** A comprehensive error reporting system will be implemented to provide users with informative error messages and to help administrators troubleshoot issues.

Page 26

9. Testing and Validation:

* **Unit Testing:** Unit tests will be written for each module to verify its functionality. A high test coverage will be aimed for.
* **Integration Testing:** Integration tests will be conducted to verify the interaction between different modules.
* **System Testing:** System testing will verify the overall functionality of the system.
* **User Acceptance Testing (UAT):** UAT will be performed by end-users to ensure the system meets their requirements.

Page 27

10. Deployment Considerations, Assumptions, and Dependencies:

* **Deployment:** The system will be deployed on a cloud platform (e.g., AWS, Azure) using containerization (Docker) and orchestration (Kubernetes).
* **Assumptions:** The system assumes the availability of a relational database (PostgreSQL) and a message queue (RabbitMQ or Kafka).
* **Dependencies:** The system depends on several third-party libraries, including Tesseract OCR, React.js, and potentially machine learning libraries. These dependencies will be managed using a package manager (e.g., npm, pip).

Page 28

11. Appendix: Detailed Database Schema

(This section would contain a detailed ER diagram and SQL schema definitions for all tables, including constraints, indexes, and data types. This is omitted here for brevity, but would be a crucial part of a real-world LLD document.)

This LLD document provides a comprehensive blueprint for the development of the Document Processing System. Remember to tailor this template to your specific requirements and the details provided in the original High-Level Design document and the `extracted_docx.docx` file (which was not provided).