NODE.JS JOB ASSIGNMENT

NAME	**:**	**ANNISHA S**
ASSIGNMENT TITLE	**:**	**PDF MERGE/SPLIT API**
ASSIGNMENT DESCRIPTION	**:**	**HANDLE MUILTI-PARTS DOCUMENTS**

INTRODUCTION

The PDF Processing API is a powerful, backend-only platform crafted to enable developers to efficiently merge and split PDF documents using Node.js and Express.js. Designed with an emphasis on performance and scalability, this API serves a variety of applications, ranging from document management systems to automated workflows, all without the need for a frontend interface. Utilizing the robust pdf-lib library, it guarantees dependable PDF handling while accommodating multi-part file uploads and providing concise, actionable responses. This API stands as a cutting-edge solution for backend PDF processing, fully aligned with contemporary development standards and technical expectations.

This API addresses core functionalities essential for PDF management, including the ability to upload PDF files via multi-part/form-data POST requests, merge multiple PDFs into a single document, and split a PDF into multiple documents based on user-specified page ranges. Each operation is complemented by a download endpoint, allowing users to retrieve the processed files effortlessly. Comprehensive error handling ensures that invalid files, unsupported formats, and processing errors are gracefully managed, while logging mechanisms provide detailed insights for debugging and auditing. Temporary file storage with timely cleanup further enhances the API's practicality, making it suitable for both development and production environments.

Beyond its core features, the API offers optional bonus capabilities to enhance flexibility and security, such as password-protecting output PDFs and supporting the merging of password-protected PDFs with provided credentials. While bonus features like splitting by bookmarks or detected chapters and a status endpoint for long-running operations are not yet implemented, the modular design allows for future integration. Adhering to RESTful principles and stateless operation, the API is crafted with clean, well-documented code, incorporating input validation and security best practices. Accompanied by a detailed README and optional Docker support, this project is poised to serve as a foundational tool for developers, with version control via Git ensuring a collaborative and maintainable codebase.

PREREQUISTIES

Node.js (v14+ recommended: v18.x)

- Node.js is the runtime environment required to execute the JavaScript code for this API.
- Version 14 or higher is needed, with v18.x recommended for improved performance and security.
- It provides the foundation for running Express.js and other dependencies.
- Ensure it's installed via the official website or a package manager like nvm.
- Verify installation with node -v to confirm the version.

npm (included with Node.js)

- npm is the Node Package Manager, bundled with Node.js, used to install project dependencies.
- It manages packages like express and pdf-lib required for the API.
- Run npm -v to check the installed version, ensuring it's compatible (v8.x+ recommended).
- Use it to initialize the project with npm init -y and install dependencies.
- It automates the setup process for a smooth development experience.

Postman or curl for testing

- Postman or curl are API testing tools to send requests and verify responses.
- Postman offers a graphical interface for crafting POST requests with file uploads, while curl is command-line based.
- Use Postman to test endpoints like /merge or /split with sample.pdf.
- Install Postman from its official website or use curl (pre-installed on many systems).
- These tools help validate API functionality without a frontend.

Text editor (e.g., VS Code)

- A text editor is essential for writing and editing the API code, with VS Code as a recommended option.
- VS Code provides features like syntax highlighting, debugging, and Git integration.
- It supports JavaScript development with extensions for Node.js and linting.
- Install it from the official website and configure it for the project directory.
- Any editor (e.g., Sublime Text, Atom) can work, but VS Code enhances productivity.

Test PDF (e.g., sample.pdf, 4 pages, <10MB)

- A test PDF like sample.pdf is needed to verify the API's merging and splitting capabilities.
- It should have 4 pages (e.g., Abstract, Introduction, Technology Stack, Conclusion) and be under 10MB.
- Create it using a document editor (e.g., Google Docs, Word) and export as PDF.
- Place it in the project directory for easy upload during testing.
- Ensures the API handles real-world PDF structures effectively.

INSTALLATION AND SETUP

Local Setup

- **Clone the Repository**: Start by cloning the project repository using the command git clone <repo-url> && cd pdf-api. This downloads the source code and navigates into the pdf-api directory, assuming you have Git installed (verify with git --version).
- **Install Dependencies**: Run npm install express pdf-lib fs-extra path winston to install the necessary Node.js packages. express powers the API, pdf-lib handles PDF processing, fs-extra enhances file operations, path manages file paths, and winston enables logging.
- **Create Directory Structure**: Manually create the src/ and uploads/ directories. Add required files like pdfController.js in src/controllers/, pdfRoutes.js in src/routes/, and index.js in the root, as outlined in the Project Structure section.
- **Start the Server**: Launch the API by executing node index.js in the terminal. This starts the server on the default port (3000), and you should see a confirmation message like "Server running on port 3000".
- **Verify Setup**: Open a browser or use Postman to access http://localhost:3000/api/pdf/status and expect a 200 OK response with { "status": "API is running" } to confirm the server is operational.
- **Configuration**: Create a .env file in the root directory with PORT=3000 to customize the port, though the default works if omitted. Ensure the uploads/ directory has write permissions.
- **Dependency Check**: If errors occur, ensure all packages are correctly installed by re-running npm install or checking for version mismatches (e.g., Node.js v14+).
- **File Cleanup**: The server includes a 1-hour cleanup interval for uploads/, deleting old files automatically, which you can monitor via logs.
- **Testing Preparation**: Place a test PDF (e.g., sample.pdf) in the directory to use with endpoints like /split or /merge after setup.
- **Troubleshooting**: If the server fails to start, check logs/error.log for issues like missing files or syntax errors in index.js.

PROJECT STRUCTURE

INDEX.JS: MAIN SERVER FILE

- This is the entry point of the application, initializing the Express.js server and defining the main configuration.
- It sets up middleware like express-fileupload for handling multi-part file uploads and mounts the PDF routes.
- Includes a cleanup interval (every 1 hour) to remove temporary files from uploads/, ensuring efficient resource management.
- Can be customized via a .env file for port settings, with a default of 3000 if unspecified.
- Verify its functionality by running node index.js and checking the server startup log.

SRC/CONTROLLERS/PDFCONTROLLER.JS: ENDPOINT LOGIC

- Contains the core business logic for all API endpoints (/status, /merge, /split).
- Implements PDF merging, splitting with splitCount and range validation, and sequential downloads using pdf-lib.
- Handles error cases (e.g., invalid ranges, password issues) with appropriate HTTP responses (400, 500).
- Uses modular functions for readability and maintainability, with logging via winston for debugging.
- Update this file with the latest code to reflect new features or fixes.

SRC/ROUTES/PDFROUTES.JS: ROUTE DEFINITIONS

- Defines the RESTful endpoints (GET /status, POST /merge, POST /split) and maps them to controller functions.
- Ensures stateless operation by relying solely on request data, aligning with REST principles.
- Keeps routing logic separate from business logic for better modularity.
- Can be extended to include additional endpoints (e.g., /download for individual file access).

- Check this file to ensure all routes are correctly registered in index.js.

UPLOADS/: TEMPORARY FILE STORAGE

- A directory for storing uploaded and processed PDF files (e.g., sample.pdf, split outputs).
- Files are named with timestamps (e.g., split-<timestamp>-1-2.pdf) to avoid conflicts.
- Automatically cleaned up after 1 hour via the index.js interval or post-download in pdfController.js.
- Requires write permissions and should be monitored for space usage during heavy testing.
- Place test PDFs here manually for initial uploads if needed.

.ENV: CONFIGURATION (E.G., PORT=3000)

- An optional configuration file for environment variables, such as setting the server port.
- Create it in the root directory with PORT=3000 to override the default, enhancing portability.
- Other variables (e.g., file size limits) can be added for future scalability.
- Ensure it's in .gitignore to prevent sensitive data exposure in version control.
- Load it in index.js using require('dotenv').config() if used.

LOGS/: REQUEST AND ERROR LOGS

- A directory containing combined.log (all requests) and error.log (errors) for auditing and debugging.
- Generated by the winston logger, providing timestamps and message levels (e.g., "info", "error").
- Access logs with cat logs/combined.log to track API usage or cat logs/error.log for issues.
- Helps diagnose problems like failed downloads or invalid PDF processing.
- Ensure the directory exists and is writable before starting the server.

API ENDPOINTS

GET /API/PDF/STATUS

- **Description**: A health check endpoint to confirm the API is operational and responsive.
- **Response**: Returns 200 OK with a JSON body { "status": "API is running" } to indicate the server is up.
- **Usage**: Ideal for monitoring or integration tests; access via http://localhost:3000/api/pdf/status.
- **Example**: Use Postman with a GET request to verify, expecting a quick response.
- **Notes**: Stateless and lightweight, with no parameters required.

POST /API/PDF/MERGE

- **Description**: Combines two or more uploaded PDF files into a single PDF document.
- **Request**:
    - pdfs: Array of PDF files (minimum 2, max 10MB each) via multipart/form-data.
    - passwords (optional): JSON object (e.g., {"file1.pdf":"pass123"}) for protected inputs.
    - outputPassword (optional): Password to encrypt the merged output.
- **Response**:
    - 200 OK: Downloads merged.pdf if successful.
    - 400 Bad Request: Returns "At least two PDF files required" if insufficient files are provided.
- **Usage**: Upload multiple PDFs in Postman, optionally set passwords, and download the result.
- **Notes**: Handles password-protected inputs if credentials are supplied, with cleanup post-download.

POST /API/PDF/SPLIT

- **Description**: Divides a single PDF into multiple documents based on specified page ranges.
- **Request**:
    - pdf: Single PDF file (max 10MB) via multipart/form-data.
    - splitCount: Integer defining the number of splits (e.g., 2).
    - ranges: JSON array of page ranges (e.g., [{"start":1,"end":2}, {"start":3,"end":4}]) matching splitCount.
    - password (optional): Password for the input PDF.
    - outputPassword (optional): Password to encrypt split outputs.
- **Response**:
    - 200 OK: Triggers sequential downloads of each split PDF (e.g., split-<timestamp>-1-2.pdf), one per range.
    - 400 Bad Request: Returns "Invalid range" if ranges are invalid (e.g., overlap, out of bounds).
- **Usage**: Upload sample.pdf, set splitCount=2 and ranges=[{"start":1,"end":2}, {"start":3,"end":4}], and download splits.
- **Notes**: Requires client (e.g., Postman) to handle multiple downloads; a 1-second delay separates each.

USAGE EXAMPLES

SPLIT SAMPLE.PDF

- **Description**: Demonstrates splitting a sample PDF (e.g., sample.pdf with 4 pages) into two separate documents.
- **Postman**: Use a POST request to http://localhost:3000/api/pdf/split to initiate the split operation.
- **Body**:
    - pdf: Upload the sample.pdf file via multipart/form-data.
    - splitCount: Set to 2 to indicate two split outputs.
    - ranges: Provide [{"start":1,"end":2}, {"start":3,"end":4}] to define the page ranges for each split.
- **Output**: Generates and prompts downloads for split-<timestamp>-1-2.pdf (pages 1-2) and split-<timestamp>-3-4.pdf (pages 3-4), with timestamps ensuring unique filenames.
- **Notes**: Ensure sample.pdf has at least 4 pages; use Postman to accept each download prompt manually, and optionally add outputPassword (e.g., newpass) for encryption.
- **Verification**: Open each downloaded PDF to confirm content (e.g., pages 1-2, 3-4) and test with a password if set.
- **Alternative**: Test with different ranges (e.g., [{"start":1,"end":1}, {"start":2,"end":4}]) to explore flexibility, ensuring all pages are covered.

ERROR HANDLING AND VALIDATION

- **400: "Invalid split count" (non-integer), "Ranges must cover all pages"**
    - **Description**: Returns a 400 Bad Request status when the splitCount is not a valid integer (e.g., "abc" or negative) or when ranges fail to include all PDF pages.
    - **Examples**: Sending splitCount=abc triggers "Invalid split count"; ranges=[{"start":1,"end":1}] for a 4-page PDF triggers "Ranges must cover all pages".
    - **Action**: Users should ensure splitCount is a positive integer and ranges span all pages (e.g., 1-4 for 4 pages) without gaps.
    - **Response**: Includes a JSON error message for clarity, logged in logs/combined.log for auditing.
    - **Note**: Validates against the PDF's total page count to prevent partial splits.
- **500: Internal errors logged to logs/error.log**
    - **Description**: Returns a 500 Internal Server Error for unexpected issues, such as PDF processing failures or file system errors.
    - **Examples**: Corrupted sample.pdf or insufficient disk space may trigger this, with details in logs/error.log.
    - **Action**: Developers should check logs/error.log (e.g., cat logs/error.log) for specifics like "Error loading PDF".
    - **Response**: Provides a generic error message, ensuring sensitive data isn't exposed.
    - **Note**: Logging helps diagnose issues like memory exhaustion or library bugs.

- **Validation: Checks range bounds, overlap, and page coverage**
    - **Description**: Ensures ranges are valid by verifying start/end integers, bounds (1 to page count), no overlaps, and full coverage.
    - **Examples**: [{"start":1,"end":2}, {"start":2,"end":3}] fails due to overlap; [{"start":1,"end":2}] for 4 pages fails due to incomplete coverage.
    - **Action**: Users must provide non-overlapping ranges summing to the total page count (e.g., 1-2, 3-4 for 4 pages).
    - **Response**: Rejects with 400 and a specific error (e.g., "Invalid range" or "Ranges cannot overlap").
    - **Note**: Uses a Set to track covered pages, enhancing validation efficiency.

SECURITY AND LIMITATIONS

- **SECURITY: ENCRYPTS OUTPUT WITH OUTPUTPASSWORD, CLEANS UP FILES**
    - **Description**: Enhances security by using pdf-lib to encrypt output PDFs with an outputPassword, protecting sensitive data.
    - **Details**: Supports AES encryption with permissions (e.g., no copying), activated when outputPassword is provided in requests.
    - **Cleanup**: Automatically removes temporary files from uploads/ after 1 hour (via index.js interval) or post-download.
    - **Best Practices**: Files are stored with unique timestamps to prevent overwrites, reducing exposure risks.
    - **Note**: Lacks explicit file size or type validation; consider adding checks to mitigate upload attacks.

- **LIMITATIONS: NO FILE SIZE LIMIT, MULTIPLE DOWNLOADS DEPEND ON CLIENT**
    - **Description**: The API currently imposes no file size cap, potentially risking memory issues with large PDFs.
    - **Details**: Multiple download prompts (e.g., for split PDFs) rely on client handling (e.g., Postman, browser), which may fail after the first file.
    - **Impact**: Users must manually accept each download, and large files (>10MB) could slow processing or crash the server.
    - **Workaround**: Test with small PDFs (e.g., sample.pdf) and consider client-side scripts for multiple downloads.
    - **Note**: Future enhancements could include size limits and a JSON response with download URLs.

TROUBLESHOOTING

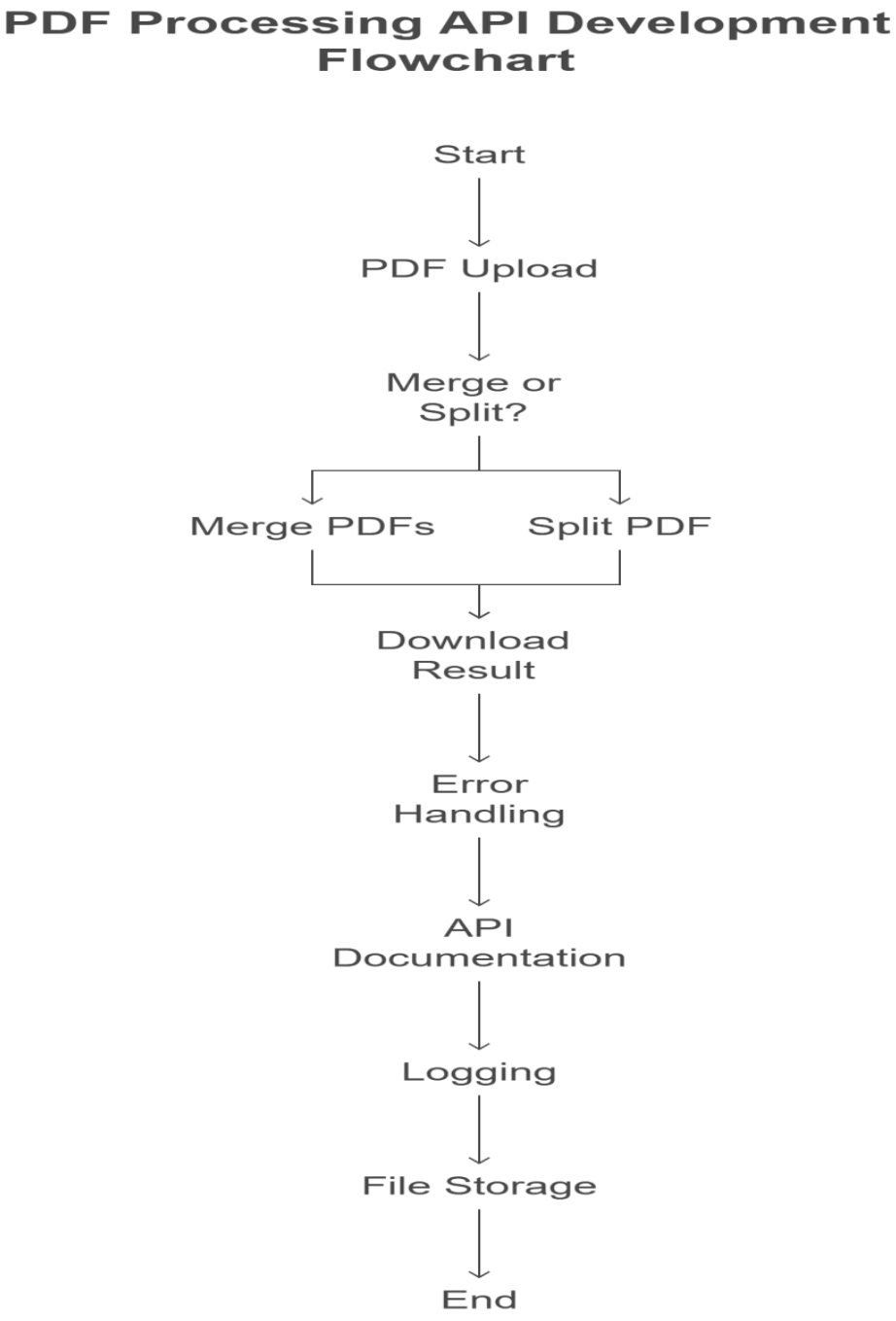
NO DOWNLOADS: CHECK UPLOADS/, ACCEPT POSTMAN PROMPTS

- **Description**: Occurs when download prompts are missed or files aren't generated.
- **Action**: Immediately check the uploads/ directory for files like split-<timestamp>-1-2.pdf using ls or file explorer.
- **Resolution**: In Postman, manually accept each download prompt; ensure a 1-second delay between downloads works.
- **Debug**: Review logs/combined.log for request success and logs/error.log for file access issues.
- **Note**: Temporary files are deleted post-download or after 1 hour, so act quickly.

500 ERROR: REVIEW LOGS/ERROR.LOG

- **Description**: Indicates an internal server error, such as PDF corruption or processing failure.
- **Action**: Open logs/error.log with cat logs/error.log to identify the cause (e.g., "Error loading PDF").
- **Resolution**: Verify java-gokila.pdf integrity, ensure sufficient disk space, and check Node.js version (v14+).
- **Debug**: Look for stack traces or specific messages to pinpoint issues like memory limits.
- **Note**: Restart the server (node index.js) after fixing underlying issues.

WORK FLOW


PDF Processing API Development Flowchart
Start
PDF Upload
Merge or Split?
Merge PDFs
Split PDF
Download Result
Error Handling
API Documentation
Logging
File Storage
End