Architecture Overview

Architecture Overview

Purpose
This document explains the high-level structure of the Document Management System. It includes an easy-to-understand description of the technologies considered and why we selected the final tech stack.

---
1. System Components
- **Frontend (React.js)**: A web application where users log in, upload documents, trigger ingestion, and view logs.
- **Backend (NestJS)**: A server application that provides APIs to handle authentication, document storage, and ingestion tracking.
- **Database (PostgreSQL)**: Stores users, uploaded documents, and ingestion logs.
- **File Storage**: Stores uploaded files either on the server (local) or on Amazon S3 (cloud storage).
- **Containerization (Docker)**: Ensures that the entire application runs consistently across all environments.

---
2. Tech Stack Considered

Backend Options:
1. **Express.js**
   - Pros: Simple, widely used, minimal setup.
   - Cons: Lacks built-in modular structure for large projects.

2. **Django (Python)**
   - Pros: Rapid development, good for data-heavy apps.
   - Cons: Less popular in Node.js ecosystem and higher latency for real-time interactions.

3. **NestJS (Chosen)**
   - Pros: Modular, scalable, built on TypeScript, easy to maintain for large projects.
   - Cons: Slight learning curve for beginners.

✅ **We chose NestJS** because it provides a well-structured, scalable architecture, is compatible with TypeScript, and makes it easy to add future microservices.

Frontend Options:
1. **Angular**
  - Pros: Strong structure, good for large enterprise apps.
  - Cons: Steeper learning curve, less flexible for small to mid projects.

2. **Vue.js**
  - Pros: Lightweight and easy to learn.
  - Cons: Smaller community compared to React.

3. **React.js (Chosen)**
  - Pros: Flexible, large community, great for reusable UI components, widely adopted.
  - Cons: Needs extra libraries for state management and routing.

✅ **We chose React.js** because it is beginner-friendly, widely used, and integrates well with modern tools like Vite, TailwindCSS, and React Query.

Database Options:
1. **MySQL**
  - Pros: Popular, easy to set up.
  - Cons: Limited support for advanced JSON data types and full-text search.

2. **MongoDB**
  - Pros: Flexible document-based database.
  - Cons: Not ideal for relational data and transactions.

3. **PostgreSQL (Chosen)**
  - Pros: Supports relational data, transactions, indexing, and advanced queries.
  - Cons: Slightly more setup needed.

✅ **We chose PostgreSQL** because it handles relational data (Users ↔ Documents ↔ Ingestion Logs) perfectly and ensures data consistency.

File Storage Options:
1. **Local Storage**
  - Pros: Fast and easy to set up.
  - Cons: Files are stored on a single server and can't scale well.

2. **AWS S3 (Chosen for production)**
  - Pros: Scalable, reliable cloud storage.
  - Cons: Requires setup and has a cost for storage/transfer.

✅ **We implemented a hybrid approach**: local storage for development and S3 for production.

Why Docker?
Docker ensures that the backend, frontend, and database run in isolated containers. This avoids the "it works on my machine" problem and makes deployment simple and repeatable.

---
3. Data Flow (Simplified)
1. A user logs in using their credentials.
2. An admin or editor uploads a document (saved locally or on S3).
3. Admin (or editor with permission) triggers ingestion.
4. Backend simulates ingestion and stores status logs.
5. User can view history or retry ingestion.

This architecture is beginner-friendly, modular, and scalable for future enhancements.