# PART-1

The Leave Management System is designed to simplify how employees and managers handle leave requests in an office environment. At its core, the system supports four key operations – adding new employees, applying for leave, approving or rejecting leave requests, and checking the leave balance of an employee.

On the backend, the system is built using FastAPI with MongoDB as the database. FastAPI provides a lightweight, fast, and asynchronous framework that makes API development seamless, while MongoDB helps in efficiently storing employee and leave details. Different API endpoints handle specific tasks, such as creating employees, submitting leave applications, validating requests, and updating leave balances. Proper error handling is also included to cover real-world edge cases, like applying for leave before the joining date, requesting more days than the available balance, overlapping leave periods, or invalid date entries. This ensures that the system behaves reliably and prevents misuse.

On the frontend, the application is structured using React in a component-based approach. Each functionality of the system, such as employee registration, leave application, leave history, and approval/rejection, is represented by a separate React component (as shown in the attached component structure). This makes the UI modular, easy to maintain, and extendable. For now, the frontend includes placeholder logic and UI elements with soothing colors, while the actual API calls can be integrated later.

By combining these two parts, the system ensures a smooth interaction where the backend enforces business rules and validations, and the frontend provides a clean interface for employees and administrators to manage leave requests effectively.

## System Description:-

1. Backend System

The backend of the Leave Management System is built with FastAPI, which provides a fast, asynchronous, and Python-friendly framework for developing RESTful APIs. The backend is responsible for handling all business logic and data validations. It connects to a MongoDB database using Motor (an async Mongo client), where employee records and leave applications are stored.

Key responsibilities of the backend include:

- **Employee Management**: Adding new employees, fetching employee details, and checking leave balances.
- **Leave Management**: Submitting new leave requests, validating dates, preventing overlaps, checking leave balance, and processing approvals or rejections.

- **Data Integrity**: Ensuring that invalid actions (like applying for leave before joining date, end date before start date, or applying for more days than available balance) are rejected with clear error messages.
- **API Endpoints**: Exposing clean, well-defined endpoints (`/employees`, `/leaves`, `/balance`, `/action`, etc.) which can be consumed by the frontend.

2. Frontend System

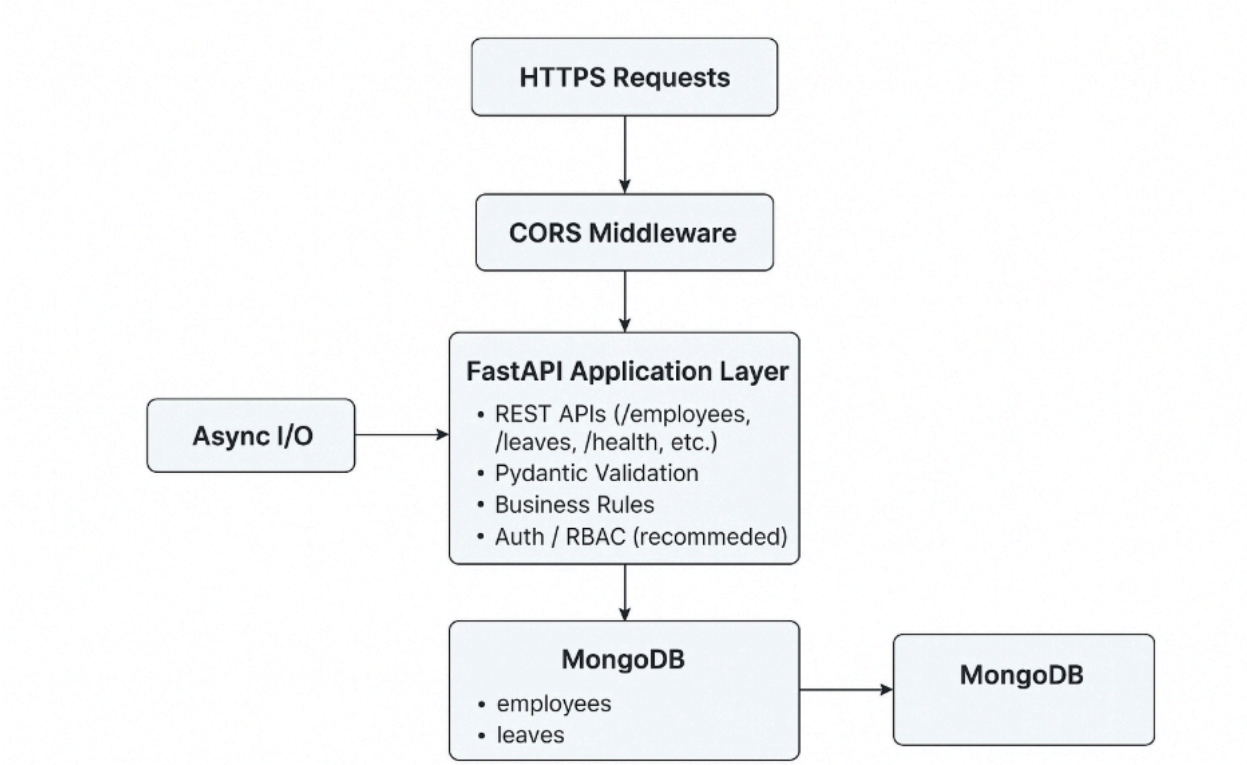
The frontend is developed using **React** with a **component-based architecture**. Each major functionality of the system is represented by a dedicated component, making the UI modular and easier to manage. For example:

- **EmployeeForm.jsx** → Add new employees
- **LeaveForm.jsx** → Apply for leave
- **ApproveRejectLeave.jsx** → Approve or reject pending requests
- **LeaveBalance.jsx** → Display leave balance for employees
- **LeaveHistory.jsx** → Show past leave applications

The frontend also has common UI elements like a **Header** and **ButtonPanel** to improve navigation and user experience. Visually, the frontend uses **light and soothing colors**, ensuring the interface looks professional and easy on the eyes. Since React is highly scalable, new features (such as dashboards, reports, or notifications) can be added later without rewriting the existing code.

# PART-2

1. High-Level Architecture Diagram




2. Interaction of DB with APIs

1. Add Employee
   Endpoint: POST /employees
   Logic:
   a) Check unique email (employees.find_one({email})).
   b) Insert employee with default leave_balance (20) and joining_date (ISO).
   DB:employees: insert_one

2. List / Get Employee
   Endpoints: GET /employees, GET /employees/{id}
   Logic:
   a) List: sort by name.
   b) Get: fetch by _id.
   DB: employees.find().sort("name", 1), employees.find_one({ _id })

3. Leave Balance
   Endpoint: GET /employees/{id}/balance
   Logic:
      a) Project only leave_balance.
   DB:employees.find_one({ _id }, { leave_balance: 1 })

4. Apply for Leave
   Endpoint: POST /leaves
   Validations (in order):
      a) Employee exists.
      b) start_date >= joining_date.
      c) end_date >= start_date.
      d) Requested days ((end-start)+1) ≤ leave_balance.
      e) No overlap with approved leaves:
   DB: leaves.insert_one({ status: "PENDING" })

5. List Leaves / Leaves by Employee
   Endpoints: GET /leaves?status=&employee_id=, GET /employees/{id}/leaves
   Logic:

      a) Flexible filtering and sort by start_date desc.

   DB: leaves.find(q).sort("start_date", -1)

6. Approve / Reject Leave
   Endpoint: POST /leaves/{leave_id}/action
   Logic:
      a) Ensure leave exists and is PENDING.
      b) If APPROVE:
         Compute days.
         Atomically decrement employee's leave_balance by days.
         Mark leave APPROVED.

      c) If REJECT:
         Mark leave REJECTED.

   DB:employees.update_one({ _id }, { $inc: { leave_balance: -days } },
      leaves.update_one({ _id }, { $set: { status: "APPROVED" } })

**Data Models:-**

The system uses two main collections in MongoDB – **employees** and **leaves** – each defined through Pydantic models in the backend.

The **Employee** model stores all information related to a staff member. Each employee has a unique identifier (`_id`) generated by MongoDB, along with their `name`, `email`, and `department`. The `email` field is validated and also enforced to be unique so that no two employees share the same email address. The model keeps track of the `joining_date`, which is the date the employee officially joined the company. Finally, every employee has a `leave_balance`, which represents the number of leave days available to them, with a default of 20 days assigned at the time of joining. This model ensures that all basic employee details are consistently recorded and makes it easy to fetch or update their leave balance.

The **Leave** model is used to manage requests for leave submitted by employees. Each leave entry has its own unique identifier (`_id`) and is tied to an employee through the `employee_id` field. A leave request contains two key dates – `start_date` and `end_date` – which define the duration of the leave period. The system automatically calculates the number of days requested based on these two dates. Every leave request also has a `status`, which can be `"PENDING"`, `"APPROVED"`, or `"REJECTED"`, depending on whether the request is waiting for review, has been accepted, or has been declined.

Additionally, there is a **LeaveAction** model, which is not stored directly in the database but is used to handle decisions on leave requests. This model has a single attribute, `action`, which can either be `"APPROVE"` or `"REJECT"`, allowing managers or administrators to update the status of a pending leave in a controlled and standardized way.

Together, these models form the backbone of the system: employees are stored with their details and leave balances, leave requests track applications and their outcomes, and actions allow controlled updates by authorized users.

3. How to Scale the System ?

- **Database Indexing** – To ensure efficiency as the dataset grows, indexes can be applied on frequently queried fields such as `employee_id`, `email`, and `status`. This significantly reduces query time and allows the system to handle larger volumes of records smoothly.

- **Pagination and Filtering in the Frontend** – Instead of rendering all employees or leave requests simultaneously, the frontend can implement pagination and filtering. This approach keeps the interface responsive and user-friendly, even when the database

contains thousands of entries.

- **Multiple Backend Instances** – As concurrent user activity increases, a single FastAPI server may not be sufficient. Running multiple backend instances behind a load balancer distributes the workload evenly and improves both system performance and reliability.

- **MongoDB Replica Sets** – For better scalability and fault tolerance, MongoDB can be deployed as a replica set. This configuration provides data redundancy and allows read operations to be spread across multiple nodes, reducing the burden on the primary database.

- **Asynchronous and Background Processing** – Tasks that do not require immediate completion, such as sending notifications or recalculating leave balances, can be offloaded to background workers. This ensures the application remains responsive, even during peak usage.

- **Caching Frequently Accessed Data** – Frequently requested information, such as leave balances or pending leave requests, can be stored in a caching system like Redis. By reducing the number of direct database queries, caching significantly improves response times.