Leave Management System API - Technical Test Documentation

Developed By

Ikram Ait Kaddi – Software Engineer

Overview

This project is a Leave Management System built using ASP.NET Core 8, implementing clean architecture and various design patterns to ensure maintainability, scalability, and testability. The API is containerized using Docker and supports deployment through docker-compose, with optional cloud deployment to Azure, Render, or Railway.

Technologies Used

- .NET 8 (ASP.NET Core Web API)
- Entity Framework Core
- SQLite
- Docker
- Swagger
- Postman

Project Structure

```
LeaveManagementSystem/
│
├── Controllers/          # API controllers (LeaveRequestsController, etc.)
│
├── DTOs/                 # Data Transfer Objects (used to decouple Models from external layers)
│
├── Mappings/             # AutoMapper profiles (DTO ↔ Model mapping configurations)
│
├── Models/               # Domain models (Employee, LeaveRequest, Enums)
│
├── Repositories/         # Interfaces and implementations for data access
│
├── Services/             # Business logic and validation services
│   ├── Validators/       # Strategy pattern for leave type validations
│   ├── Report/           # Report generation services (Factory Pattern)
```

```
│   ├── Services & Implementations /
│
├── Data/               # EF Core DbContext
│   └── ApplicationDbContext.cs
│
│
├── Utilities/          # Custom utilities (e.g., PredicateBuilder, helpers, config loader)
│   └── Logging/        # Singleton logger (optional if logging is custom)
│   └──Seeders/         # Seeder classes for initial data population
│
├──Migrations/
│
├── leave.db            # SQLite database file (can be mounted in Docker volume)
├── Program.cs          # Main entry point and configuration
├── appsettings.json    # Configuration settings
├── Dockerfile          # Docker build config for the app
└── docker-compose.yml  # Compose file for app + database setup
```

Running the Project Locally

1. Clone the repository
2. Open in Visual Studio
3. Run with Docker or:

```
dotnet run
```

API will be available at: http://localhost:8080/swagger

Database

SQLite is used and seeded on app start with test data.

Day 1 – Setup & Modeling

• Implemented models:
  - Employee: Id, FullName, Department, JoiningDate
  - LeaveRequest: Id, EmployeeId, LeaveType, StartDate, EndDate, Status, Reason, CreatedAt
• Created CRUD endpoints for LeaveRequest.
• Used DTOs and AutoMapper for better data encapsulation and mapping.

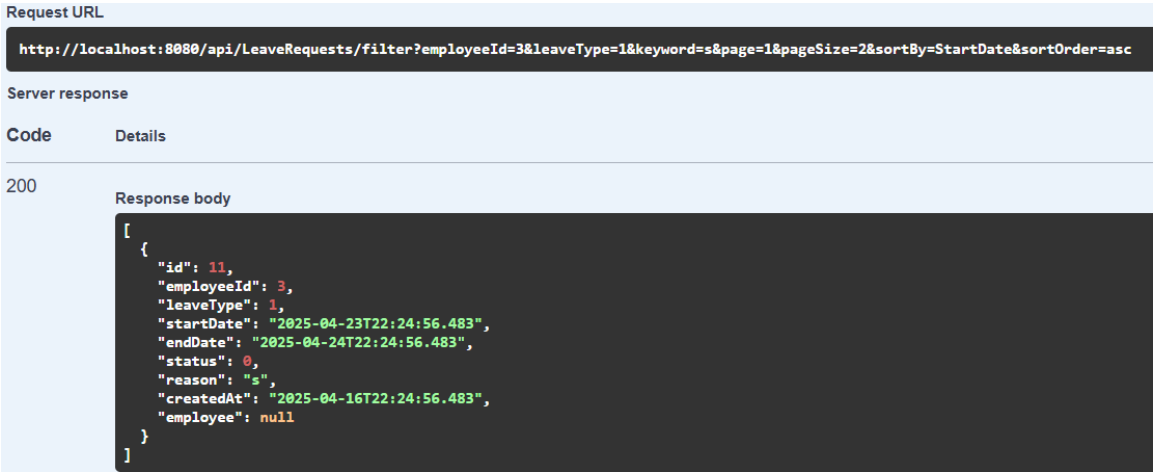
- API Endpoints
  - Leave Requests

Method	Endpoint	Description
GET	/api/LeaveRequests	Get all leave requests
GET	/api/LeaveRequests/{id}	Get leave request by ID
POST	/api/LeaveRequests	Create leave request
PUT	/api/LeaveRequests/{id}	Update leave request
DELETE	/api/LeaveRequests/{id}	Delete leave request

Day 2 – Filtering & Business Logic

- Implemented filtering endpoint: GET /api/leaverequests/filter with support for:
  - employeeId, leaveType, status, startDate, endDate, keyword in reason

  Added:
  - Pagination (page, pageSize)
  - Sorting (sortBy, sortOrder)
  - Search functionality



- ***Business Rules Implementation***

The Leave Management System incorporates essential business constraints to ensure valid and controlled leave processing. These rules are handled within the **application service layer** to keep the controller thin and maintain separation of concerns.

- **No overlapping leave dates per employee**:
  Before creating or approving a leave request, the system checks whether the

employee already has an overlapping request. This ensures accurate tracking of leave periods.

- **Maximum 20 annual leave days per year**:
  The system calculates the total number of approved *Annual* leave days taken by an employee in the current year. If the new request would exceed the 20-day limit, it's rejected.
- **Sick leave requires a non-empty reason**:
  To ensure accountability, sick leave requests must include a justification. The system enforces this as a validation rule.

```
curl -X 'POST' \
  'http://localhost:8080/api/LeaveRequests' \
  -H 'accept: text/plain' \
  -H 'Content-Type: application/json' \
  -d '{
  "id": 21,
  "employeeId": 3,
  "leaveType": 0,
  "startDate": "2025-04-17T22:18:26.177Z",
  "endDate": "2025-04-17T22:18:26.177Z",
  "status": 0,
  "reason": "string",
  "createdAt": "2025-04-17T22:18:26.177Z",
  "employee": {
    "id": 3,
    "fullName": "ikram ait kaddi",
    "department": "it",
    "joiningDate": "2025-04-17T22:18:26.177Z"
  }
}'
```

Request URL

```
http://localhost:8080/api/LeaveRequests
```

Server response

Code	Details
400 *Undocumented*	Error: Bad Request Response body `Leave request is not valid according to the business rules.`

Day 3 – Reporting & Deployment

• Implemented Summary Report endpoint: GET /api/leaverequests/report?year=YYYY with optional filters (department, date range).

- Json Format:

- Csv Format:

• Added Admin Approval Endpoint: POST /api/leaverequests/{id}/approve with validation (only Pending requests can be approved).

Request URL

```
http://localhost:8080/api/LeaveRequests/20/approve
```

Server response

Code	Details
200	Response body

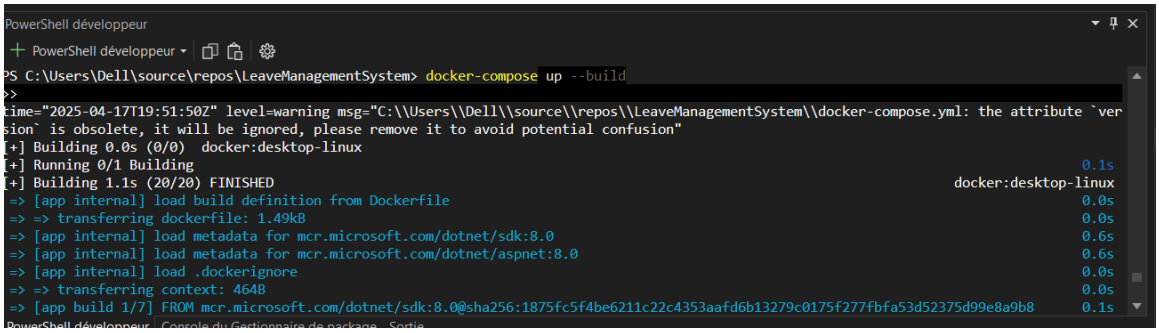
```
{
  "id": 20,
  "employeeId": 3,
  "leaveType": 0,
  "startDate": "2025-04-17T22:18:26.177",
  "endDate": "2025-04-17T22:18:26.177",
  "status": 1,
  "reason": "string",
  "createdAt": "2025-04-17T22:18:26.177",
  "employee": null
}
```

Deployment

• Created Dockerfile for the API and a docker-compose.yml file to run API and SQLite together.

• Dockerfile created in the root directory with multi-stage build.
• docker-compose.yml manages app and database service (SQLite is file-based and not a separate server like PostgreSQL or SQL Server).
• Volume mounted for SQLite to allow persistent data.
• Application tested via Postman using exposed port 8080.

- Run docker-compos up –build:



• Ensured app is accessible via http://localhost:8080 in Docker.

Design Patterns Used (Referenced in Code Comments)

Throughout the code base, several software design patterns were applied to ensure modularity, maintainability, and scalability. Each pattern is briefly documented in the relevant classes or services via comments for clarity:

- **Repository Pattern**
  Abstracted database access logic using interfaces (`IRepository<T>`) and concrete implementations. This makes the code loosely coupled and testable, supporting clean architecture principles.
- **Strategy Pattern**
  Used to encapsulate **leave validation logic** for different leave types.
  Each type of leave (Annual, Sick, Other) has its own validation strategy implementing a common interface.
  For example:
  - `AnnualLeaveValidator` checks the 20-day limit.
  - `SickLeaveValidator` enforces a non-empty reason.
  - `DefaultLeaveValidator` handles generic validation for other types.

  The system selects the appropriate strategy at runtime based on the `LeaveType`. This promotes **open/closed principle**: new validation types can be added without modifying existing logic.

- **Factory Pattern**
  Used in report generation to dynamically select output format (e.g., JSON or CSV) by returning the appropriate report generator class based on a parameter. This avoids switch/case logic in the controller and makes the report system extensible.
- **Singleton Pattern**
  Used for managing shared configuration settings or services that should only be instantiated once (e.g., logging or centralized app settings).

Postman Collection

Includes all APIs, filters, approval, and reports.
Import `LeaveManagementSystem.postman_collection.json` into Postman.

Conclusion

This project demonstrates proficiency in .NET backend development including REST API design, filtering, pagination, applying business logic, Docker containerization, and implementing design patterns. The solution is extensible and follows clean architecture principles, making it suitable for enterprise-level applications.