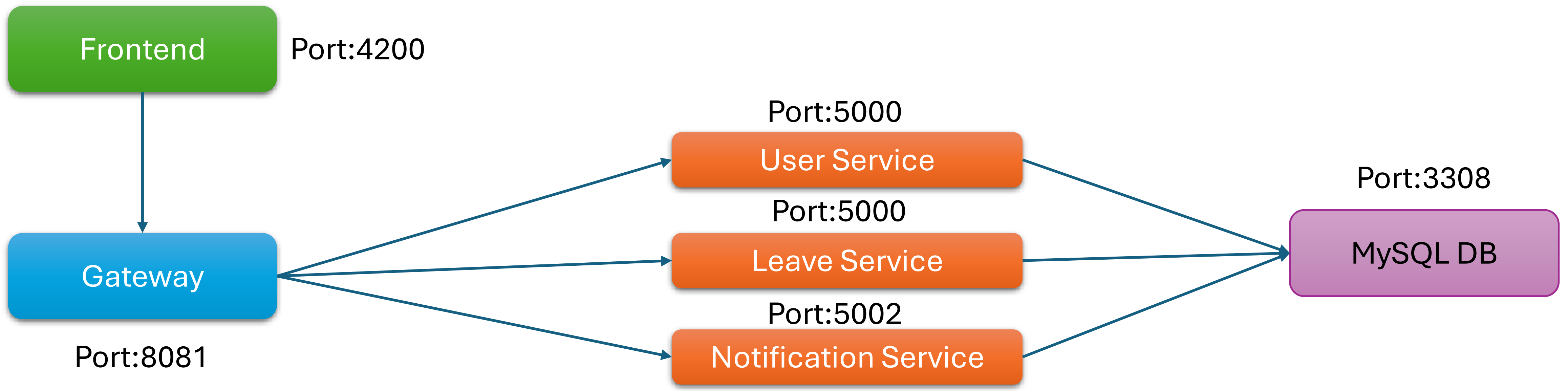
Employee Leave Management System

- A Microservices-based system for handling employee leaves, approvals, and notifications.
- Built with **Python (Flask REST Api services)**, **Angular frontend**, **MySQL DB**, and deployed with **Docker containers**.

Architecture Style

- Microservices-based architecture
- Independent services communicating via REST APIs
- Services: User Service, Leave Service, Notification Service, Gateway, Frontend (Angular), MySQL DB

Architecture Diagram



Docker Compose manages containerized deployment.

User Service

- Manages user registration, login, and roles (Employee/Manager).
- Ensures secure password hashing and JWT authentication.
- Provides role-based access control (managers can approve leaves, employees cannot).
- Handles profile updates and user validations.
- APIs exposed:
  - POST /users → Register new user
  - GET /users → List all users
  - POST /login → Authenticate user and issue JWT

Leave Service

- Central service for leave request creation and approval.
- Implements leave types rule (Medical, Sick, Privileged).
- Prevents duplicate leave applications for the same date range.
- Tracks leave history for auditing/reporting.
- Integrates with User Service for employee/manager validation.
- APIs exposed:
  - POST /leave → Apply leave
  - GET /leave/{id} → Get leave status
  - PUT /leave/{id}/approve → Approve leave request
  - PUT /leave/{id}/reject → Reject leave request
  - GET /leave/history/{user_id} → Fetch user leave history

Notification Service

- Handles real-time notifications for approvals, rejections, and reminders.
- Sends alerts via email/SMS/portal messages (configurable).
- Works asynchronously to avoid slowing down core services.
- Stores notification logs for auditing.
- APIs exposed:
  - POST /notify → Trigger notification
  - GET /notify/status/{id} → Check notification status
  - GET /notify/history/{user_id} → View all past notifications

Gateway Service

- Single entry point for frontend and external clients.
- Provides API aggregation — clients don't need to call services separately.
- Handles load balancing across multiple service instances.
- Manages security (CORS, authentication checks, request validation).
- Improves scalability by decoupling frontend and backend services.
- Routes requests to:
  - User Service
  - Leave Service
  - Notification Service

Containerization

- All microservices are Dockerized for consistent environments.
- Containers allow easy scaling (e.g., multiple leave-service instances).
- Each service has its own image and can be updated independently.
- Shared Docker network (lms_net) ensures communication between containers.
- Volumes used for persistent MySQL storage.
- Images are lightweight.

Deployment with Docker Compose

- docker-compose.yml defines multi-container setup.
- Provides service dependency management (depends_on ensures DB starts first).
- Supports health checks to verify service availability.
- Maps ports between host and containers (e.g., MySQL: 3308 → 3306).
- Developers can run the entire stack using:
  - docker compose up -d –build
- Can be extended to Kubernetes or cloud platforms later.

Summary

- Scalable Microservices Architecture → Each service is modular and independently deployable.
- Role-based Access Control (RBAC) → Secure separation of employee and manager responsibilities.
- API-first Design → RESTful APIs make it easy to integrate with third-party systems.
- Containerized Deployment → Simplifies setup, ensures environment consistency.
- Centralized Database (MySQL) → Provides reliable and persistent data storage.
- Asynchronous Notifications → Keeps users informed without blocking main operations.
- Gateway as API Aggregator → Improves security, load handling, and client simplicity.
- Extensible → Future additions like reporting and ML-based leave predictions.
- Business Impact → Improves leave management efficiency, reduces manual errors, and enhances employee satisfaction.