مدينة زويل للعلوم والتكنولوجيا
Zewail City of Science and Technology

- **Title**: LearnVibe LMS

- **Subtitle**: Scalable Microservices-Based Learning Management System

- **Presented by**:
  - sandy mohamed 202202034
  - hesham ashraf 202201477
  - Pakinam khaled 202202233
  - Sama reda 202202246

What is LearnVibe?

- LearnVibe is a full-stack Learning Management System (LMS) built using microservices.
- Designed for modern online education: scalable, resilient, and secure.

Key Functionalities:

- Course & user management
- Video content streaming
- Secure login via Google OAuth2
- Redis caching for performance
- RabbitMQ for async background tasks

Tech Stack Highlights:

- Backend: Go (Golang), Gin
- Database: PostgreSQL
- Caching: Redis
- Messaging: RabbitMQ
- Storage: MinIO
- Testing : Grafana/k6

Key Services:

- **API Gateway** – Auth + request routing
- **CMS Service** – Manages courses/users
- **Content Service** – Manages file/video access
- **PostgreSQL** – Two databases for isolation
- **Redis** – Caching for faster access
- **RabbitMQ** – Async background messaging
- **MinIO / Azure Blob** – File storage

Trade-offs Analysis:

- **Microservices Architecture**
  - **Pros**: Modularity, scalability
  - **Cons**: Complexity in integration, requires robust orchestration
- **OAuth2 (Google) Authentication**
  - **Pros**: Secure, trusted external service
  - **Cons**: Dependency on external service (Google)

- **RabbitMQ for Messaging**
    - **Pros**: Scalable, handles async tasks effectively
    - **Cons**: Additional infrastructure overhead
- **MinIO / Azure Blob Storage**
    - **Pros**: Flexible storage solution
    - **Cons**: Requires setup of proxy or additional configurations

Challenges Faced and Solutions Implemented:

- **Scalability Issues**
    - Challenge: Handling high traffic in peak times
    - Solution: Load balancing, database sharding
- **Security Challenges**
    - Challenge: Securing Google OAuth2 authentication tokens
    - Solution: Implementing token encryption and secure token storage

- **Performance Bottlenecks**
    - Challenge: Slow response times under load
    - Solution: Caching with Redis, optimizing database queries
- **Async Task Management Issues**
    - Challenge: Delays in processing background tasks
    - Solution: Optimized message queuing, retry strategies in RabbitMQ
- **Integration Issues**
    - Challenge: Service communication between microservices
    - Solution: Standardizing REST API interfaces, improving error handling

Performance Evaluation:

- **Performance Testing Framework**
    - Tools: Grafana, k6
- **Key Metrics**
    - Response Time: 200ms under normal load, 600ms under heavy load

- Throughput: 1000 requests per second
- Redis Cache Hits: 98% hit rate

- RabbitMQ Message Processing Time: 50ms

- **Real-World Performance Results**

  - During a stress test with 5000 concurrent users, the system handled 90% of requests within 300ms, with minimal impact on system performance.

Functional Requirements

User Authentication & Authorization

- Users can register and log in via Google OAuth2.
- Role-based access control (Admin, Instructor, Student).
- Users can update their profile (name, email, profile picture).

Course Management

- Instructors can create, update, delete, and publish courses.
- Students can browse, enroll, add,and drop courses.
- Course materials (videos, PDFs, quizzes) can be uploaded and accessed. -Support for course prerequisites (students must complete Course A before Course B).
-Instructors can schedule live sessions (Zoom/Google

Meet integration).

Enrollment & Payment

- Users can enroll in free or paid courses.
- Automatic invoicing and receipt generation.
- Support for refunds and cancellations.

Recommendation System

- AI-based personalized course recommendations based on user behavior.

Notification System

-Real-time notifications for:

- Enrollment confirmation
- Course updates
- Assignment deadlines
- Payment confirmations

-Support for email, in-app, and push notifications.

Dashboard & Analytics

-Students can track course progress (completion percentage, quiz scores).

-Instructors can view student engagement metrics.

- Admins can monitor platform analytics (user growth, course popularity, system health).

Non-Functional Requirements

Performance

- The API must handle 10,000 requests per second (RPS) at peak load.
- The database should support 20,000 queries per second (QPS) using read replicas.
- Video streaming should support 2,500 concurrent users with low latency.
- The system should provide a response time of less than 500ms for 95% of requests.

Scalability

The system should support 100,000 registered users with 5,000 concurrent users.

- Auto-scaling must be implemented for API and database services to handle traffic spikes.
- The system should allow dynamic resource provisioning based on traffic load.
- A Content Delivery Network (CDN) should be used to optimize static content deliver

Availability

- The system must ensure 99.9% uptime, limiting downtime to 8.76 hours/year.
- The application should support graceful degradation, ensuring core functionalities remain operational during failures.
- Redundant instances of critical services should be deployed to prevent single points of failure

Resilience

- Circuit breakers & retry mechanisms should be implemented to handle failures in service-to-service communication.
- Fallback strategies should be designed for critical services to prevent complete system failure.
- A distributed message queue (e.g., Kafka, RabbitMQ) should be used for asynchronous processing (e.g., notifications, payments).
- Database replication and sharding should ensure failover support and high availability

Security

- OAuth2-based authentication using Google Login must be implemented for secure access.
- Role-based access control (RBAC) must restrict functionalities for Admins, Instructors, and Students.
- Sensitive user data (e.g., passwords, payment details) must be encrypted at rest and in transit using AES-256 and TLS 1.2+.
- The system should comply with GDPR and ISO 27001 standards for data protection.
- Web Application Firewall (WAF) should be used to prevent security threats such as DDoS attacks and SQL injection

Maintainability

- Centralized logging using OpenSearch should be implemented for debugging and security monitoring.
- Microservices should be loosely coupled to allow independent updates and deployments.

- The system must follow a modular architecture, allowing easy replacement or enhancement of components.

Observability

- Application metrics (e.g., API latency, database queries, active users) should be collected using Prometheus/Grafana.

- Distributed tracing should be implemented to track requests across microservices (e.g., Jaeger, Zipkin).

- Alerting mechanisms should be in place to notify administrators of critical failures.

Deployment & CI/CD

- Automated CI/CD pipelines should be used for deployments to minimize manual errors.

- Rolling updates should be used to prevent downtime during deployment.

- Blue-Green Deployment should be considered for major releases to ensure rollback capability**.**

Storage Requirements

- Each course requires an average of 5GB of storage (videos, documents).
    - With 10,000 courses, total storage required = 50TB, hosted on cloud storage (e.g., AWS S3, GCS).
    - 90% of video traffic should be served via CDN, reducing direct storage bandwidth.
    - Database backup and recovery policies should ensure data protection and minimize data loss.

User Stories

As an Admin, I want to:

- Manage users (approve instructors, suspend accounts) so that I can ensure platform security.
- Monitor platform usage (active users, enrollments, payments) so that I can maintain system performance.
- View reports and analytics so that I can track student engagement and instructor activity.
- Manage system settings (payment configurations, email notifications) so that I can customize platform behavior.
- Moderate content (course reviews, discussion forums) so that I can prevent spam or inappropriate content.

As an Instructor, I want to:

- Create, edit, and delete courses so that I can provide learning content to students.

- Upload course materials (videos, PDFs, quizzes) so that students can access structured learning resources.
- Schedule live sessions so that I can interact with students in real time.
- Track student progress so that I can provide personalized support.
- Send announcements so that I can notify students about course updates.
- View student engagement metrics (time spent, quiz scores) so that I can improve course effectiveness.

As a Student, I want to:

- Search and browse courses so that I can find relevant learning opportunities.

- Enroll in a course (free or paid) so that I can access learning materials.

- Pay for premium courses so that I can unlock additional content.
- Receive personalized course recommendations so that I can discover relevant learning paths.
- Track my progress (completed lessons, quiz scores) so that I can measure my learning.
- Receive notifications (new lessons, deadlines, announcements) so that I stay updated.

- Engage in course discussions so that I can ask and answer questions.
- Download certificates after course completion so that I can showcase my learning achievements.

As a System (LMS), I want to:

-Authenticate users via Google OAuth2 so that login is secure and seamless.

- Store user roles and permissions so that access control is enforced.

- Process payments securely so that transactions are safe.
-Send automated notifications so that users are informed about updates.
- Analyze user behavior so that course recommendations are personalized.
- Ensure high availability (scalability and resilience) so that the system runs smoothly under heavy load.

Use Cases

Use Case 1: User Registration & Authentication

- **Use Case ID:** UC-01
- **Actors:** Student, Instructor, Admin
- **Preconditions:** User must have a Google account.

● **Basic Flow:**

○ The user clicks the "Sign Up/Login" button.

○ The system redirects the user to Google OAuth2 for

authentication.

○ The user selects a Google account to sign in.

○ The system verifies the user’s identity and creates a session.

○ The system assigns a role (Admin, Instructor, or Student) based on user data.

○ The user is redirected to the respective dashboard.

● **Alternative Flow:**

○ **A1:** If authentication fails, the system displays an error message and allows the user to retry.

● **Postconditions:** The user is authenticated and redirected to

their dashboard.

Use Case 2: Course Creation

- **UseCaseID:**UC-02
- **Actors:**Instructor

Preconditions: The instructor must be logged in.

- **Basic Flow:**
  - The instructor clicks the "Create Course" button.
    - The system displays a course creation form.
    - The instructor enters the course details (title, description, etc.).
    - The system validates the input.
    - The instructor uploads course materials (videos, documents).
    - The system stores the files and links them to the course.
    - The instructor clicks "Publish."

○ The system makes the course visible for enrollment.

- **Alternative Flow:**
  - **A1:** If validation fails, the system prompts the instructor to correct errors before proceeding.
  - **A2:** If file upload fails, the system notifies the instructor to retry.
- **Postconditions:** The course is available for enrollment.

Use Case 3: Course Enrollment

- **Use Case ID:** UC-03
- **Actors:** Student
- **Preconditions:** The student must be logged in.
- **Basic Flow:**
  - The student searches for a course.
  - The system displays a list of available courses.
  - The student clicks "Enroll" for a selected course.

- The system checks if the course is free or requires payment.
- If the course is paid, the student completes the payment process.
- The system processes the payment and confirms enrollment.
- The student receives a confirmation notification.
- The course is added to the student's dashboard.

● **Alternative Flow:**

- **A1:** If payment fails, the system notifies the user and provides retry options.
- **A2:** If course capacity is full, the system displays a waitlist option.

- **Postconditions:** The student gains access to the course.

Use Case 4: Payment Processing

- **Use Case ID:** UC-04
- **Actors:** Student
- **Preconditions:** The student selects a paid course.
- **Basic Flow:**
  - The student selects a premium (paid) course.
  - The system redirects the student to the payment page.
  - The student enters payment details.
  - The system validates and processes the transaction.
  - The student confirms the payment.
  - The system adds the course to the student's dashboard.
  - The system records the transaction in the database.

- The student receives a payment confirmation email.

- **Alternative Flow:**

- **A1:** If payment is declined, the system notifies the user and prompts for alternative payment methods.

- **A2:** If the transaction times out, the system provides an option to retry.

- **Postconditions:** The student is enrolled, and the payment is recorded.

Architectural Drivers (Quality Attributes)

Scalability

The system ensures scalability by using a Content Delivery Network (CDN) to optimize content delivery, auto-scaling mechanisms to dynamically adjust resources based on demand, and load balancing to distribute traffic efficiently across multiple instances.

Resilience

To maintain system reliability, the application implements retry mechanisms to handle transient failures, circuit breakers to prevent cascading failures,and caching to reduce load on backend services and improve response times.

Security

Security is enforced through OAuth2-based authentication for secure user login, Role-Based Access Control (RBAC) to restrict access based on user roles, encryption for data protection, and rate limiting to mitigate potential abuse or DDoS attacks.

Performance

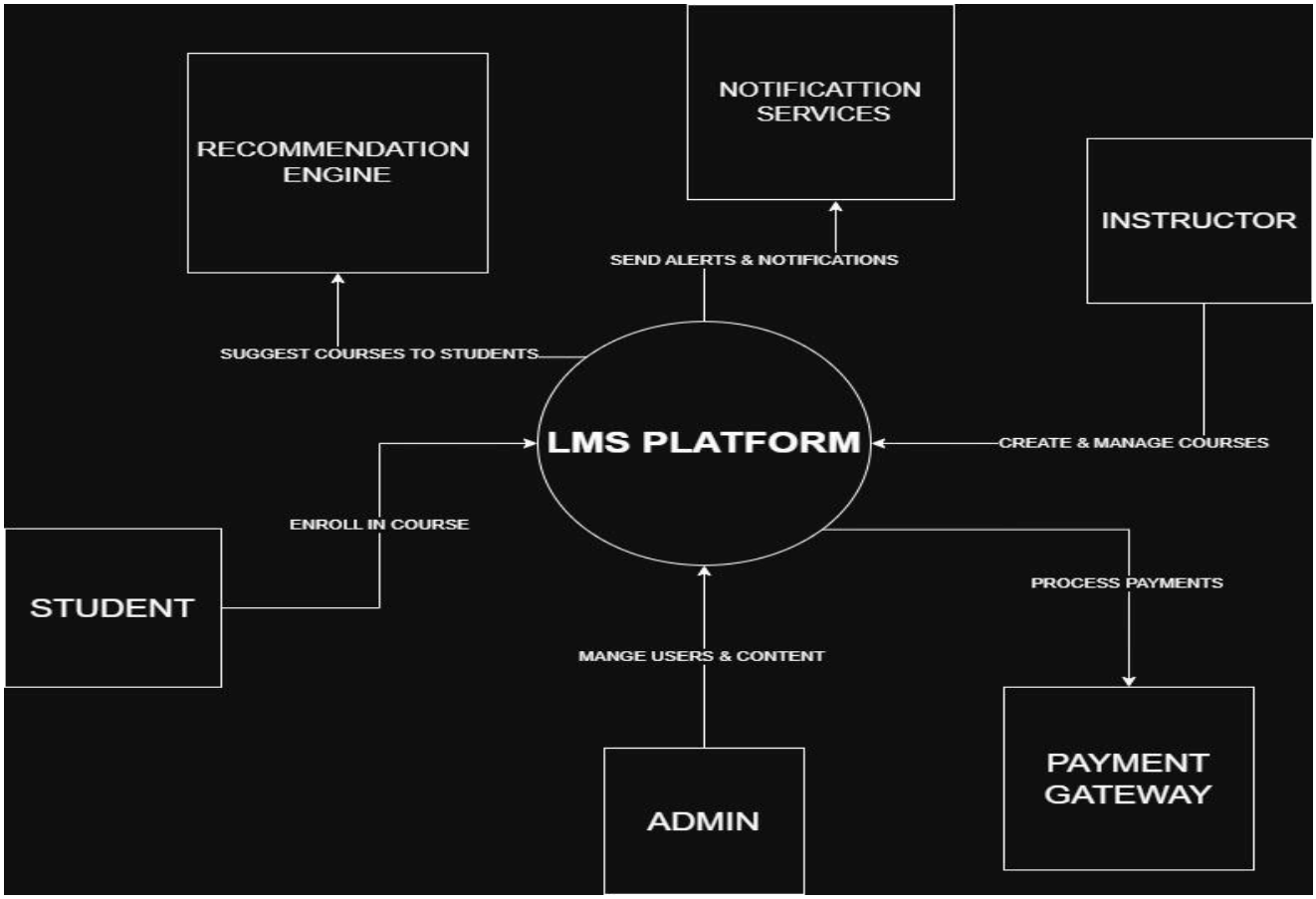
The system optimizes performance by utilizing caching to reduce redundant database queries, asynchronous processing for background tasks (e.g., notifications, report generation), and optimized database queries to enhance data retrieval efficiency.

Observability

For monitoring and observability, the application integrates OpenSearch for centralized logging, Prometheus for collecting system metrics, and Grafana for real time visualization and alerting to detect potential issues proactively.

Software Architecture

Context diagram : This diagram highlights the high-level interactions between the LMS and its surrounding entities, which include students, instructors, administrators, external services, and system components**.**



1. Student

- Interaction: Students interact with the LMS to enroll in courses.
- Role: They are the primary users of the platform, accessing learning materials, participating in assessments, and tracking their progress.
- Flow:
    - Students enroll in available courses through the LMS interface.
    - They receive course recommendations from the platform based on their interests and activity.

2. Instructor

- Interaction: Instructors create and manage courses using the LMS.
- Role: They develop course content, upload learning materials, and manage assessments and grades.
- Flow:
    - Instructors use the LMS tools to create new courses, update existing ones, and interact with students.

3. Admin

- Interaction: Admins manage users and content.
- Role: Responsible for system-level management including user roles, platform settings, content moderation, and access control.
- Flow:
    - Admins oversee the overall operation of the LMS and ensure it runs smoothly.

4. Recommendation Engine

- Interaction: The LMS suggests courses to students using a Recommendation Engine.
- Role: This component analyzes user behavior, preferences, and past activity to recommend suitable courses.
- Flow:
    - LMS sends student data to the engine, which returns a list of recommended courses.

5. Notification Services

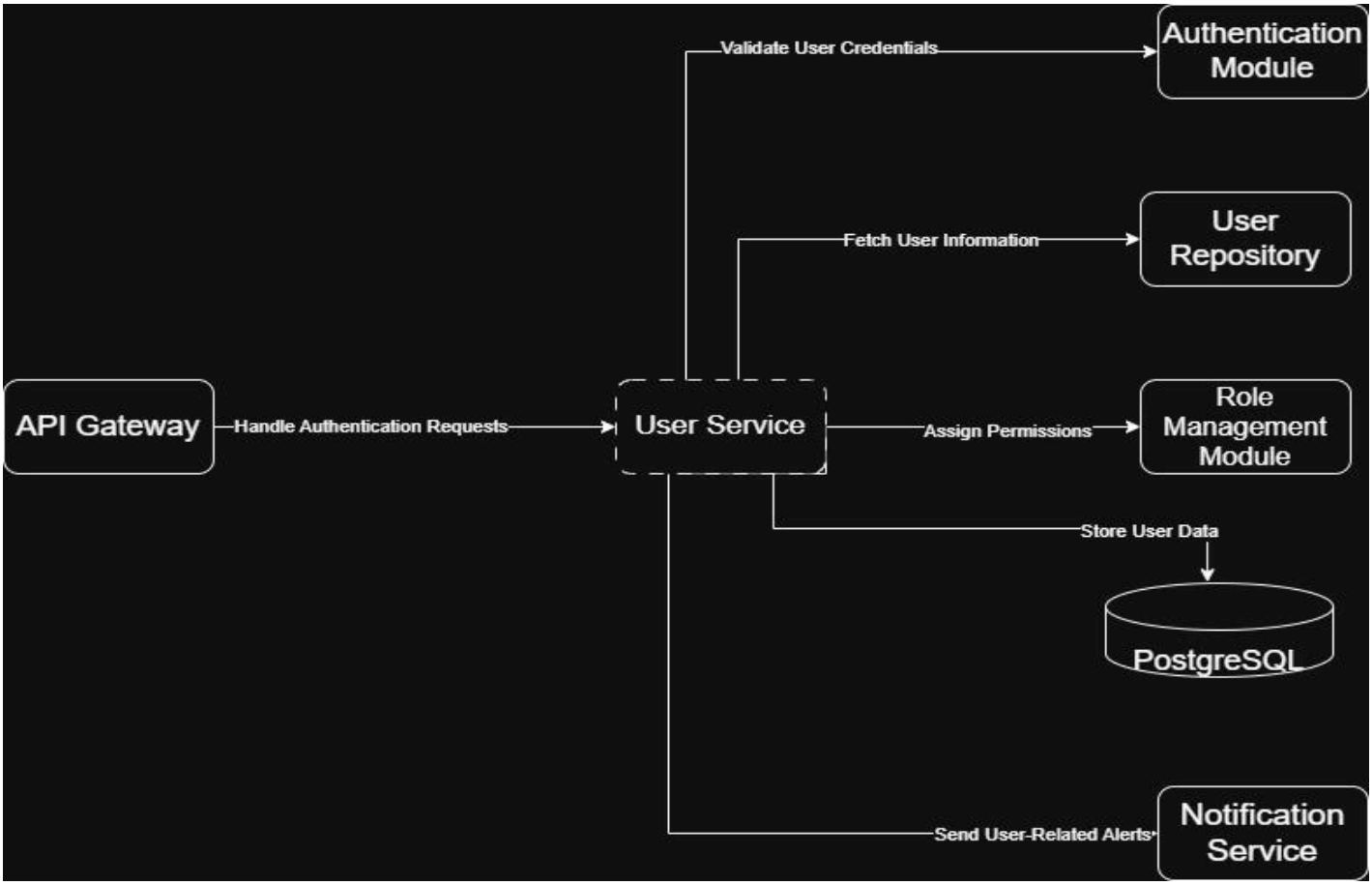
- Interaction: LMS uses Notification Services to send alerts and notifications.
- Role: Keeps users informed about updates, deadlines, announcements, and messages.
- Flow:
    - LMS triggers notifications (e.g., course updates, reminders) to be delivered to users via email, SMS, or in-app messages.

6. Payment Gateway

- Interaction: Handles payment processing for course enrollments.
- Role: Facilitates secure financial transactions between students and the LMS.
- Flow:

- LMS forwards payment details to the gateway, which processes the transaction and returns the status.

Component diagram: This diagram highlights the key modules that interact with the User Service and the direction of data and control flow between them.



Central Component: User Service

- **Role**: The User Service is responsible for managing user-related operations such as authentication, authorization, data storage, and notifications.
- **Functionality**:
    - Handles incoming authentication requests.
    - Coordinates with other modules to validate, store, and manage user data.
    - Assigns roles and permissions to users.
    - Sends notifications related to user activities.

1. API Gateway

- **Interaction**: Sends requests to the User Service.
- **Role**: Acts as the single entry point for all client requests. It routes user authentication and management requests to the User Service.
- **Data Flow**:
    - The API Gateway handles authentication requests and forwards them to the User Service for processing.

2. Authentication Module

- **Interaction**: Receives credential validation requests from the User Service.
- **Role**: Verifies the credentials (e.g., username and password) of users trying to log in.

- **Data Flow**:
    - The User Service sends credentials for validation.
    - The Authentication Module returns authentication results.

3. User Repository

- **Interaction**: Accessed by the User Service to retrieve user profile information.
- **Role**: Stores existing user information such as personal details, preferences, and status.
- **Data Flow**:
    - The User Service fetches user information as needed (e.g., during login or profile updates).

4. Role Management Module

- **Interaction**: Receives permission assignment requests from the User Service.
- **Role**: Manages user roles and access rights (e.g., admin, student, instructor).
- **Data Flow**:
    - The User Service assigns permissions based on user roles during registration or profile updates.

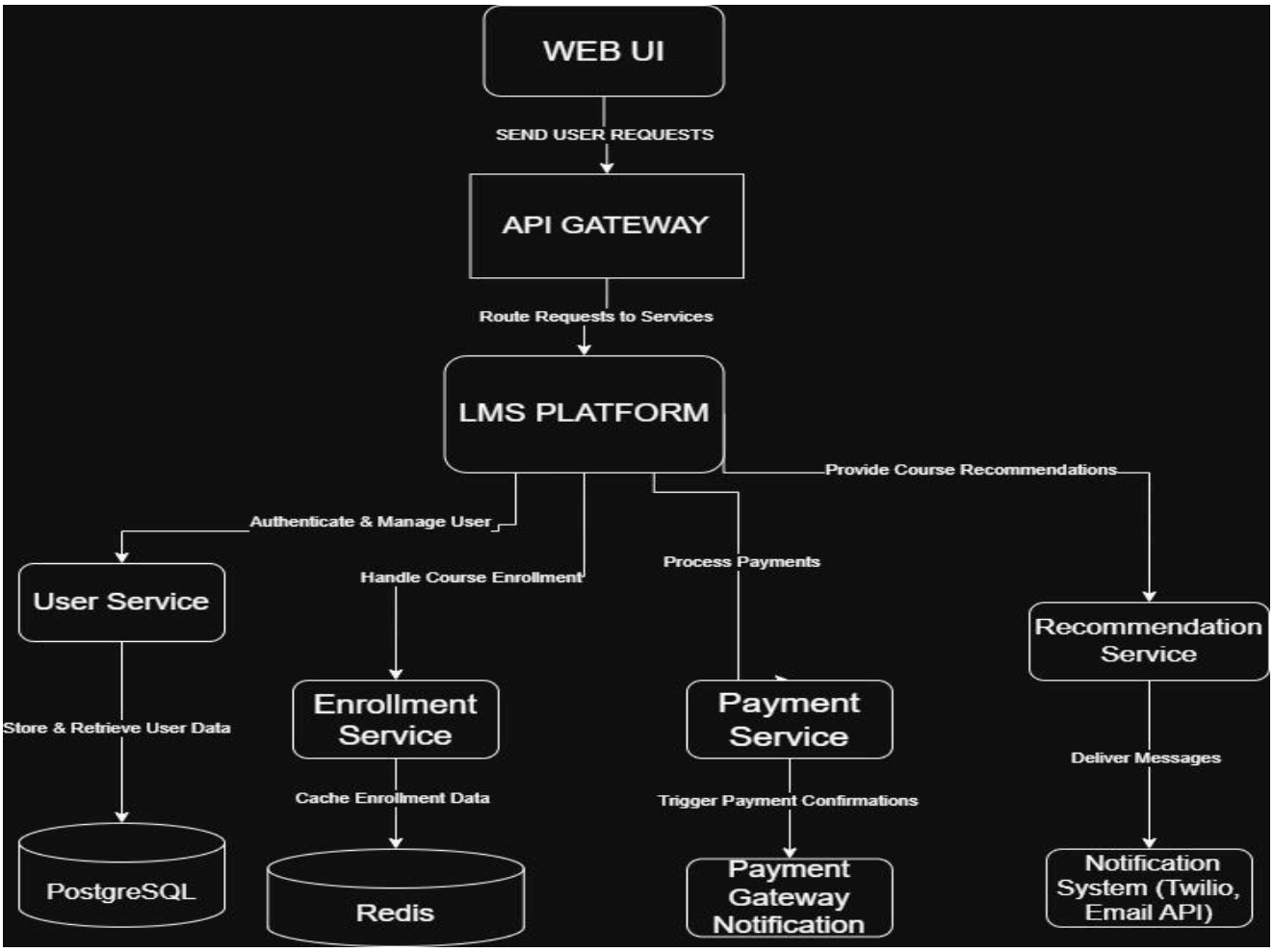
5. PostgreSQL

- **Interaction**: Used by the User Service to store or update user data.
- **Role**: Acts as the main relational database where persistent user data is stored.
- **Data Flow**:
    - The User Service stores user data (e.g., new registrations, updates to user profiles).

6. Notification Service

- **Interaction**: Receives alert/notification triggers from the User Service.
- **Role**: Sends emails, messages, or alerts to users for actions like registration confirmation, password reset, or security alerts.
- **Data Flow**:
    - The User Service sends user-related alerts to the Notification Service to be delivered to the end user.

Container diagram: This diagram highlights the high-level architecture of the LMS by showing how the system is structured into containers (applications, services, databases, etc.) and how they interact. Each container represents an executable or deployable unit that plays a distinct role within the system.



1. Web UI (Frontend Application)

- **Purpose**: Acts as the interface through which users (students, instructors, admins) interact with the system.

- **Responsibilities**:

    - Sends user actions (e.g., enroll, pay, browse) to the backend.

    - Renders dynamic content such as courses, dashboards, and user profiles.

- **Technology Stack Example**: React.js, Angular, or Vue.js.

2. API Gateway

- **Purpose**: Serves as the entry point for all client requests.
- **Responsibilities**:
    - Routes requests to the appropriate backend services.
    - Performs cross-cutting concerns like authentication, rate limiting, and logging.
- **Role in System**: Decouples the client from direct access to services, enhancing modularity and security.

3. LMS Platform (Application Backend)

- **Purpose**: Core application that orchestrates and manages the learning platform's operations.
- **Responsibilities**:
    - Handles main business logic.
    - Coordinates requests between various services (user, payment, enrollment, recommendation).
- **Function**: Central processing unit of the system; delegates responsibilities to specialized services.

Supporting Backend Services

4. User Service

- **Purpose**: Manages user-related operations such as authentication, registration, and profile updates.
- **Responsibilities**:
    - Store and retrieve user data.
    - Authenticate users.
- **Connected To**:
    - **PostgreSQL** database for persistent user data storage.

5. Enrollment Service

- **Purpose**: Handles course enrollment processes for users.
- **Responsibilities**:
    - Manage enrollment and withdrawal operations.
    - Track and cache enrollment data.
- **Connected To**:
    - **Redis** in-memory database for fast caching and real-time updates.

6. Payment Service

- **Purpose**: Processes user payments for paid courses or subscriptions.

- **Responsibilities**:
    - Initiate and verify transactions.
    - Confirm payments via third-party payment gateway notifications.
- **Connected To**:
    - **Payment Gateway Notification** system or webhook confirmations.

7. Recommendation Service

- **Purpose**: Provides personalized course suggestions based on user activity, preferences, or history.
- **Responsibilities**:
    - Analyze user behavior or history.
    - Send recommended courses to users.
- **Connected To**:
    - **Notification System** to deliver course recommendations.

8. Notification System

- **Purpose**: Sends alerts and transactional messages to users.
- **Responsibilities**:
    - Deliver payment confirmations, course updates, or personalized messages.
- **Technologies Used**:
    - **Twilio** for SMS notifications.

- **Email APIs** (like SendGrid, Mailgun) for email notifications.

Databases & Caching

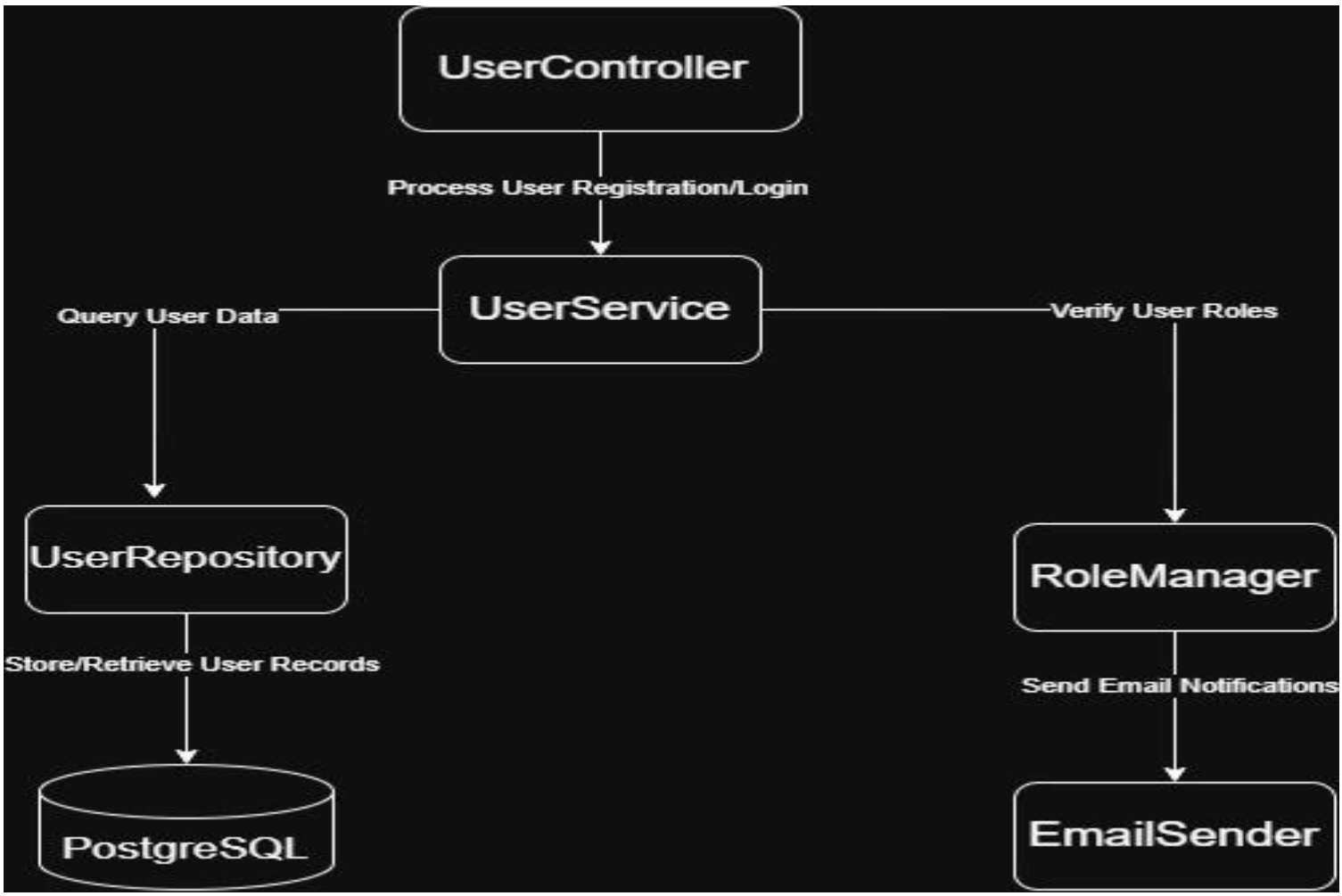
PostgreSQL

- **Used By**: User Service.
- **Stores**: Persistent user records and profile data.

Redis

- **Used By**: Enrollment Service.
- **Stores**: Temporary enrollment information for fast access and efficient user experience.

Code diagram: This diagram highlights the flow of user registration and login functionality in a modular backend application. It follows a layered and service-oriented architecture that ensures separation of concerns, reusability, and maintainability.



1. UserController

- **Role**: Acts as the entry point for handling HTTP requests related to user operations (e.g., registration, login).

- **Responsibility**: Receives the user input from the frontend or API calls and forwards it to the `UserService` for processing.

- **Interaction**: Sends user credentials and actions to `UserService`.

2. UserService

- **Role**: Serves as the business logic layer for user-related operations.
- **Responsibilities**:
    - Processes requests from the `UserController`.
    - Queries user data using the `UserRepository`.
    - Verifies user roles via the `RoleManager`.
    - Coordinates interactions between multiple modules.
- **Interaction**:
    - Talks to the `UserRepository` to access user data.
    - Calls `RoleManager` to verify if the user has appropriate permissions.
    - Triggers email notifications via the `EmailSender` if needed.

3. UserRepository

- **Role**: Data access layer (DAO - Data Access Object).
- **Responsibilities**:
    - Abstracts the direct interaction with the PostgreSQL database.
    - Performs CRUD operations (Create, Read, Update, Delete) on user records.

- **Interaction**: Connects with `PostgreSQL` to store or retrieve user data.

4. PostgreSQL

- **Role**: Relational Database Management System (RDBMS).
- **Responsibility**: Stores user records persistently.
- **Interaction**: Receives SQL queries from `UserRepository`.

5. RoleManager

- **Role**: Role verification and access control.
- **Responsibilities**:
    - Determines user roles and access permissions (e.g., admin, member).
    - Ensures users perform only actions permitted by their roles.
- **Interaction**:
    - Responds to role verification requests from `UserService`.
    - Triggers `EmailSender` for sending role-based notifications.

6. EmailSender

- **Role**: Communication service for email notifications.
- **Responsibilities**:
    - Sends emails such as registration confirmation, password reset, or role change alerts.

- **Interaction**: Called by `RoleManager` to notify users based on role-related events.