Course Recommendation System

Date: 29 Sept 2024

Introduction

This document aims to provide a comprehensive overview of the design and architecture of the Course Recommendation System. It details the software components, design considerations, and the development approach for a system designed to offer personalized course recommendations to users based on their preferences and historical data.

System Overview

The Course Recommendation System is designed to interact with users through a web interface, allowing them to view, search, and enroll in courses that are recommended based on their interests and past activities. The system integrates a backend developed in Django, which handles data management, user authentication, and recommendation logic, with a frontend developed in React, providing a responsive user experience.

Design Considerations

Assumptions and Dependencies:

- **Assumptions:**
  - Users will have continuous internet access to interact with the system.
  - The recommendation engine's accuracy is dependent on the availability of sufficient historical user data.
- **Dependencies:**
  - The system relies on the Django framework for the backend and React for the frontend.
  - Database operations assume the availability of a SQLite/PostgreSQL server.

General Constraints

- The system must operate under the constraint of handling multiple users concurrently without degradation of performance.

Goals and Guidelines

- **Goals:**
    - To provide accurate course recommendations that enhance the learning experience.
    - To ensure the system is scalable and maintainable.
- **Guidelines:**
    - All code must adhere to the DRY principle to avoid redundancy.
    - The system should be easy to update and maintain.

Development Methods

- The project will utilize Agile development methodologies with two-week sprints, allowing iterative and incremental development.

Architectural Strategies

Strategy 1: Microservices Architecture

- The backend will be divided into microservices, each handling a specific part of the system's functionality (courses, users, recommendations).

Strategy 2: API-First Development

- Development will prioritize the creation of RESTful APIs to ensure that the system's services are modular and can be easily accessed by the React frontend.

System Architecture

The system is divided into several key components, each responsible for a segment of functionality:

Component 1: User Management

- Handles user registration, authentication, and profile management.

Component 2: Course Management

- Manages all aspects of course data, including creation, modification, and retrieval.

Component 3: Recommendation Engine

- Generates personalized course recommendations using machine learning algorithms based on user data.

Detailed System Design

Module 1: User Authentication and Management

This module is responsible for user-related functionalities, including registration, login, password management, and user profile updates.

Key Features:

- **Registration**: Allows users to create an account using email and password. It validates email formats and ensures password strength.
- **Login**: Provides authentication mechanisms using email and password, along with OAuth integrations (e.g., Google, Facebook) for social login.
- **Session Management**: Manages user sessions with support for secure cookies and token-based authentication (JWT).
- **Password Reset**: Users can reset their password via email verification.
- **Profile Management**: Users can update their personal details, preferences, and view their course history.

Data Flow:

- User registration inputs are validated on the frontend.
- Upon successful validation, data is sent to the backend for processing and persistence in the database.
- JWT tokens are generated for login sessions and passed to the frontend for authenticated requests.

Module 2: Course Management

This module handles the creation, storage, updating, and retrieval of course-related data.

Key Features:

- **Course List**: A comprehensive list of available courses.
- **Course Details**: Provides detailed information about each course.

Data Flow:

- The course catalog is fetched from the PostgreSQL database and displayed in the frontend for browsing.
- When a user enrolls in a course, the backend updates the database with enrollment details and returns confirmation to the frontend.

Scalability:

- Courses can be served via a microservice-based architecture, where the course management service is independent of other services. This allows for scaling the service independently to handle spikes in user activity (e.g., during course enrollment periods).

Module 3: Recommendation Engine

The recommendation engine is the core module responsible for generating personalized course recommendations for users.

Key Features:

- **Personalized Recommendations**: The engine uses OPENAI to recommend courses based on a user's interaction history, preferences, and performance in previous courses.
- **Collaborative Filtering**: Suggests courses by comparing the user's activity with similar users. This ensures that courses popular among users with similar preferences are recommended.
- **Content-Based Filtering**: Recommends courses by analyzing course content (e.g., subjects, topics) and matching them to the user's learning history and interests.

Data Flow:

1. **Data Collection**: User interactions (e.g., course enrollments) are logged and stored in the database.
2. **Recommendation Delivery**: Recommended courses are displayed on the user's dashboard and updated based on user behavior and preferences.

Scalability and Performance:

- The recommendation engine is isolated as a microservice, allowing it to scale independently when handling large amounts of data or high user traffic.

Architectural Pattern Comparison

1. Architectural Pattern Comparison: Monolithic vs. Microservices

Monolithic Architecture

- **Overview**: Initially, the system used a monolithic architecture where all components (frontend, backend, database) resided in a single codebase.
- **Performance**: The system, using SQLite, supported around 50 concurrent users before performance degradation.
- **Scalability**: Limited, as the entire system needed to be scaled as a whole.

Microservices Architecture

- **Overview**: PostgreSQL was deployed as a microservice, allowing for independent scaling and isolated failures. The system’s components (user management, recommendation engine, course management) were separated into distinct services.
- **Performance**: After migrating to PostgreSQL in a microservices architecture, the system handled over 75 concurrent users with improved stability.
- **Scalability**: The microservice setup enabled independent scaling of specific services, significantly improving the system's ability to handle increased load.

Aspect	Monolithic	Microservices
Scalability	Limited	High (independent service scaling)
Fault Isolation	Low (failure in one part affects all)	High (isolated failures)
Maintainability	Difficult with large codebase	Easier (small, independent services)

2. Containerization vs. Virtualization

Containerization (Docker)

- **Setup**: The system was containerized using Docker, with separate containers for the Django backend and PostgreSQL database.
- **Performance**: Faster service startup times (~5 seconds) and efficient resource utilization. Handled over 100 concurrent users smoothly.
- **Advantages**: Lightweight, portable, and easy to deploy across environments.

Virtualization (Virtual Machines)

- **Setup**: The system was also tested using VirtualBox VMs running Django and PostgreSQL in separate virtual machines.
- **Performance**: Slower startup times (~50-60 seconds) with higher resource consumption. Supported fewer users (~60-90) before performance degradation.
- **Disadvantages**: VMs are resource-intensive due to each instance requiring its own OS and virtualized hardware, leading to slower performance compared to containers.

Aspect	Containers (Docker)	Virtualization (VMs)
Startup Time	Fast (~5 seconds)	Slow (~50-60 seconds)
Resource Efficiency	High (shared OS kernel)	Low (each VM requires its own OS)

3. Cloud Computing vs. Non-Cloud Setup

Cloud Setup (AWS)

- **Overview**: The system was deployed on AWS (**t3a.medium**) using Amazon ECS for container orchestration and RDS for managed PostgreSQL.
- **Performance**: Handled 100+ concurrent users without any significant latency spikes due to auto-scaling provided by AWS. AWS RDS optimized PostgreSQL performance, and the system benefited from AWS's managed infrastructure for improved reliability and scaling.
- **Benefits**: Scalability, automatic resource management, and cloud-based monitoring via CloudWatch.

Non-Cloud Setup (Local)

- **Overview**: The system was run locally using Docker. Performance started to degrade with around 100 concurrent users, and scaling required manual intervention.
- **Challenges**: Limited scalability, no automated monitoring, and resource management required local hardware upgrades for better performance.

Aspect	Cloud Setup (AWS)	Non-Cloud (Local)
Scalability	Auto-scaling with AWS ECS	Manual scaling
Performance	Consistent under high load	Degrades under heavy load

Glossary

- **API** (Application Programming Interface): A set of protocols for building and integrating application software.
- **Backend**: Server-side components of a computing architecture, typically handling data storage and processing.
- **CRS** (Course Recommendation System): Refers to the system designed to provide course recommendations to users.
- **Django**: A high-level Python web framework that encourages rapid development and clean, pragmatic design.
- **React**: A JavaScript library for building user interfaces, particularly for single-page applications where you need fast interaction.