Software Engineering Project-3: TeamSync

Team-29

Contents

Git repo link

Github repo link

Task1: Requirements and Subsystem

1 Functional and Non-functional Requirements

1.1 Functional Requirements

1.1.1 User Management

- **FR1.1**: The system shall allow users to register and authenticate with a role of either *Student* or *Admin*.
- **FR1.2**: The system shall allow students to view their registered courses and teams on a personalized dashboard.

1.1.2 Course Management

- **FR2.1**: The system shall allow admins to create courses and define the maximum team size for each.
- **FR2.2**: The system shall allow admins to delete a course, which recursively deletes all associated teams and tasks.
- **FR2.3**: The system shall allow students to register or withdraw from courses.
- **FR2.4**: Withdrawing from a course shall automatically remove the student from any associated team.

1.1.3 Team Management

- **FR3.1**: The system shall allow students to:
  - Create a team in a registered course.
  - View all existing teams of that course and their members.
  - Request to join a team only if the student is not a part of any team of that course.
  - Invite students to join their team.
  - Accept invitations, which moves them from one team to another after confirmation.
  - View students not in any team.
  - Exit a team, notifying other teammates.
- **FR3.2**: The system shall automatically delete a team if its size reaches zero.

1.1.4 Notification System

- **FR4.1**: The system shall provide real-time notifications for:
  - Join requests.
  - Team invitations.
  - Team member exits.
- **FR4.2**: Notifications shall include custom messages and interactive options (Accept/Decline).

1.1.5 Task Management

- **FR5.1**: The system shall allow team members to create, edit, and delete tasks.
- **FR5.2**: The task view shall include:
  - Task name and description.
  - Completion checkbox.
  - Responsible team member.
  - Option to take responsibility.

1.2 Non-Functional Requirements

1.2.1 Security

- **NFR1.1**: Only team members shall have access to view and modify their team's tasks.
- **NFR1.2**: The system shall implement role-based access control to ensure that users can only perform actions permitted by their role (Admin or Student).

1.2.2 Data Integrity

- **NFR2.1**: The system shall ensure **atomicity** during critical operations such as course deletion and team switching—either all related data is updated or none.
- **NFR2.2**: The system shall guarantee **consistency** by maintaining valid states after transactions (e.g., no student can be in multiple teams per course).
- **NFR2.3**: The system shall provide **isolation** for concurrent operations, such that simultaneous actions (e.g., two students accepting an invite to a team with only one slot) are handled safely without data corruption or race conditions.
- **NFR2.4**: The system shall ensure **durability** by persisting all committed transactions in permanent storage.

1.2.3 Usability

- **NFR3.1**: Notifications for invitations and join requests shall include actionable buttons (e.g., "Accept", "Decline") to streamline user decisions and interactions.
- **NFR3.2**: User flows (e.g., team creation, task assignment) shall require minimal clicks and clear feedback messages for user actions.

1.3 Architecturally Significant Requirements

1.3.1 NFR1.3

Why Architecturally Significant:

- **NFR1.3** demands **role-based access control (RBAC)** across the system, influencing service boundaries and possibly middleware layers.

1.3.2 Data Integrity Requirements (NFR2.1 NFR2.2)

Why Architecturally Significant:

- These affect **transaction management** and **data consistency layers**.
- **NFR2.1 (Atomicity)** and **NFR2.2 (Consistency)** require careful **transactional design**, especially with cascading deletes (e.g., when deleting courses).

1.3.3 Team Membership Constraints (FR3.1 - FR3.2)

Why Architecturally Significant:

- Enforcing "one team per course per student" and dynamic team deletion requires **tight coupling of business logic with the data model**.
- Impacts database schema design (e.g., enforcing unique constraints and foreign keys).
- Requires **event-driven architecture or cascading logic** to trigger side-effects like automatic team deletion.

1.3.4 Course and Team Deletion Logic (FR2.2)

Why Architecturally Significant:

- Recursive delete operation must maintain integrity across **multiple related entities** (courses → teams → tasks).
- Impacts **database schema, referential integrity constraints, and cascading logic** in services.
- Involves both transactional safety and performance considerations.

1.3.5 Usability-Driven Actionable Notifications (NFR3.1, FR4.2)

Why Architecturally Significant:

- Impacts UI/UX architecture and how the frontend communicates with backend actions.
- Requires **real-time update propagation** and **state synchronization** between client and server.
- Tightly coupled with the notification system and real-time infrastructure.

2 SubSystem Overview

There are 4 subsystems in our system,

- User Management Subsystem.
- Team Management Subsystem.
- Notification Subsystem.
- Task Management Subsystem.

2.1 User Management Subsystem

- **User Roles**: Admin, Student.
- **Signup & Sign-in System**: User authentication & session management.
- **Student Dashboard**: View registered courses & teams.

2.2 Team Management Subsystem

- **Admin Functionalities**:
  - Create course and set max team size.
  - Delete course (Recursive: Delete teams in it, tasks in team).
- **Student Functionalities**:
  - Register/Withdraw from courses (Automatically exits the team if in one).
  - For every registered course:
    - Create a new team.
    - View existing teams & their members.
    - Request to join existing teams.
    - Invite other students to join their team (even if the student is a member of any team) (but not a teammate).

    - When accepting another team's invitation: Automatically exits current team after confirmation and then joins new team.
    - View students who are not in any team.
    - Exit from a team (automatically notifies other teammates).
    - If team size = 0: Delete team.

- **Constraints**:
  - A **student can only be in one team per course**.
  - **Before accepting an invitation**, check if the **team is full**.
  - **Before accepting a request to join**, check if the **student is already in a team**.

2.3 Notification Subsystem

- **Real-time notifications for**:
  - Requests to join a team.
  - Invitations to join a team.
  - Team members leaving a team.
- **Notifications template**:
  - Custom messages.
  - Accept/Decline buttons.

2.4 Task Management Subsystem

- Create, edit, delete tasks & subtasks.
- **Task template**:
  - Name, description, task completion checkbox.
  - Displays responsible team members.
  - Take responsibility option.
- **Constraints**:
  - Only team members can see the team tasks.

Task2: Architecture Framework

3 Stakeholder Identification

Stakeholders, Concerns, and Viewpoints (IEEE 42010)

Stakeholder	Concerns	Viewpoints Addressing Concerns	View
Students	• Easy team formation and task management • Notification of invites and team updates • Usable and intuitive UI	User Experience, Information	Registered Course view and Teams view
Admins	• Create/manage courses and set rules • View and delete course data • Maintain consistency when deleting courses or teams	Functional, Structural	Data Flow View
	• Ensure smooth operation for many users	Performance, Scalability	Deployment View, Load Testing Results View
Developers	• System should follow design principles and best practices • Should be maintainable and modular	Development, Structural	Component and Class Diagram View
System Architects	• High-level system design with modularity and scalability • Alignment with software architecture patterns	Architectural, Structural	Layered Architecture View, Component Interaction View

4 Architectural Decision Records (ADRs)

4.1 ADR 1: Decision to Use Microservices Architecture Over Monolithic Architecture

Stakeholders: Developers, System Architects
Status: Accepted
Context: The project required a scalable and maintainable architecture to handle the following needs:

- **Scalability:** Ability to handle a growing number of students, teams, and courses without performance degradation.
- **Modularity:** Ability to make changes to one subsystem without impacting others.

Two options were considered:

- **Monolithic Architecture:** A single, unified codebase for all functionalities.
- **Microservices Architecture:** Separate, independent services for functionalities like user management, team management, notification.

Decision: We chose the **Microservices Architecture** because:

- It allows **independent scaling** of subsystems like notifications and real-time updates, which may require higher computational resources.
- It enables **modular development and deployment**, improving maintainability and reducing downtime during updates.

Consequences:

- **Positive:**
  - Increased scalability.
  - Allows different teams to work on different services independently.
- **Negative:**
  - Added complexity in managing microservices, requiring tools like service discovery and API gateways.
  - Potential for latency issues due to inter-service communication.

—

4.2 ADR 2: Merging Team and Course Subsystems into a Single Microservice

Stakeholders: Developers
Status: Accepted
Context: The project requires frequent interaction between **team management** and **course management** subsystems. Key drivers for this decision include:

- **High Coupling:** Teams are inherently tied to courses (e.g., students register for courses and form teams in those courses).
- **Performance:** Maintaining these as separate microservices would result in frequent cross-service communication, increasing load on the server and introducing potential latency.
- **Simplification:** Combining them reduces the complexity of inter-service communication.

Decision: We decided to merge the **Team Management** and **Course Management** subsystems into a single microservice to minimize coupling and improve system performance.
Consequences:

- **Positive:**
  - Reduced server load by eliminating frequent cross-service API calls.
  - Simplified database schema and improved transaction consistency (e.g., course deletion propagates to team deletion seamlessly).
  - Easier to maintain due to centralized logic.
- **Negative:**
  - Slightly reduced modularity. Any changes to one subsystem may require testing of the other.
  - Increased codebase size for the merged microservice.

—

4.3 ADR 3: Choice of PostgreSQL Over MongoDB

Stakeholders: System Architects, Developers
Status: Accepted
Context:
The project requires a database solution to store structured data, including user roles, courses, teams, and tasks. The following drivers were considered:

- **Data Model:** The data has clear relationships (e.g., teams belong to courses, students belong to teams).
- **Future Scalability and Portability:** The system may need to migrate to another SQL-based database (e.g., MySQL, Oracle) in the future.

Two options were considered:

- **PostgreSQL:** A relational database with ACID compliance, advanced querying capabilities, and wide compatibility with SQL standards.
- **MongoDB:** A NoSQL database offering high scalability and flexibility for unstructured data but limited support for relational operations.

Decision:
We chose **PostgreSQL** because:

- **Relational Data Support:** PostgreSQL is well-suited for relational data models, supporting foreign keys, constraints, and joins to ensure data integrity and enforce relationships.

Consequences:

- **Positive:**
  - Supports easier migration to other SQL-based databases, ensuring future portability and interoperability.
- **Negative:**
  - Schema changes may require more effort compared to NoSQL solutions.

4.4 ADR 4: Adoption of Layered Architecture

Stakeholders: Developers, System Architects
Status: Accepted
Context: The project required a clean and modular architecture to promote maintainability and separation of concerns. Key drivers included:

- **Code Organization:** Clearly defining responsibilities for data access, business logic, and user interaction.
- **Maintainability:** Easier to update individual layers without affecting others.
- **Scalability:** Supporting horizontal scalability by segregating database, logic, and user-facing components.

The chosen layered architecture follows this format:

$$Database \rightarrow Model \rightarrow DAO(DataAccessObject) \rightarrow Resource \rightarrow Frontend$$

Decision: We adopted a **Layered Architecture** because:

- It enforces **separation of concerns**, with each layer having a distinct responsibility.
- The DAO layer abstracts database access, ensuring that business logic (Resource) is independent of the persistence layer.
- The frontend is decoupled from backend services, allowing UI development to proceed independently.

Consequences:

- **Positive:**
  - Improves code readability and maintainability.
  - Facilitates testing at each layer independently.
  - Simplifies future migrations, like switching databases, as the DAO layer abstracts data access.
- **Negative:**
  - May introduce slight overhead due to additional abstraction layers.

Task3: Architectural Tactics and Patterns

5 Architectural Tactics

5.1 Performance → Resource Management (Concurrency Optimization)

Explanation: To improve concurrency, Database access is distributed across specialized microservices (e.g., user, team, notification services), which reduces contention and improves parallelism. This modular design ensures efficient handling of multiple user requests, particularly for real-time notifications and team management. By isolating responsibilities and optimizing resource usage, the system minimizes latency and maintains high throughput even under heavy load.

Addresses:

- **Non-Functional Requirement (NFR3.1):** Real-time responsiveness for notifications.
- Supports scalability and consistent performance under high user load.

5.2 Security → Resisting Attacks (Authenticate Users)

Explanation: User authentication ensures that only legitimate users can access the system. Techniques like secure session tokens (JWT) are used. This protects the system against unauthorized access.

Addresses:

- **NFR1.2:** Enforces secure session management for single-device login.
- **NFR1.3:** Supports role-based access control by verifying user identity.

5.3 Security → Resisting Attacks (Authorize Users)

Explanation: Role-based access control (RBAC) restricts users from performing actions beyond their roles (e.g., Admins managing courses vs. Students managing tasks).

Addresses:

- **NFR1.1:** Ensures only team members can access and modify their team's tasks.
- **NFR1.3:** Supports secure operations by enforcing role-specific privileges.

5.4 Usability → Design Time (Separate UI from the Rest of the System)

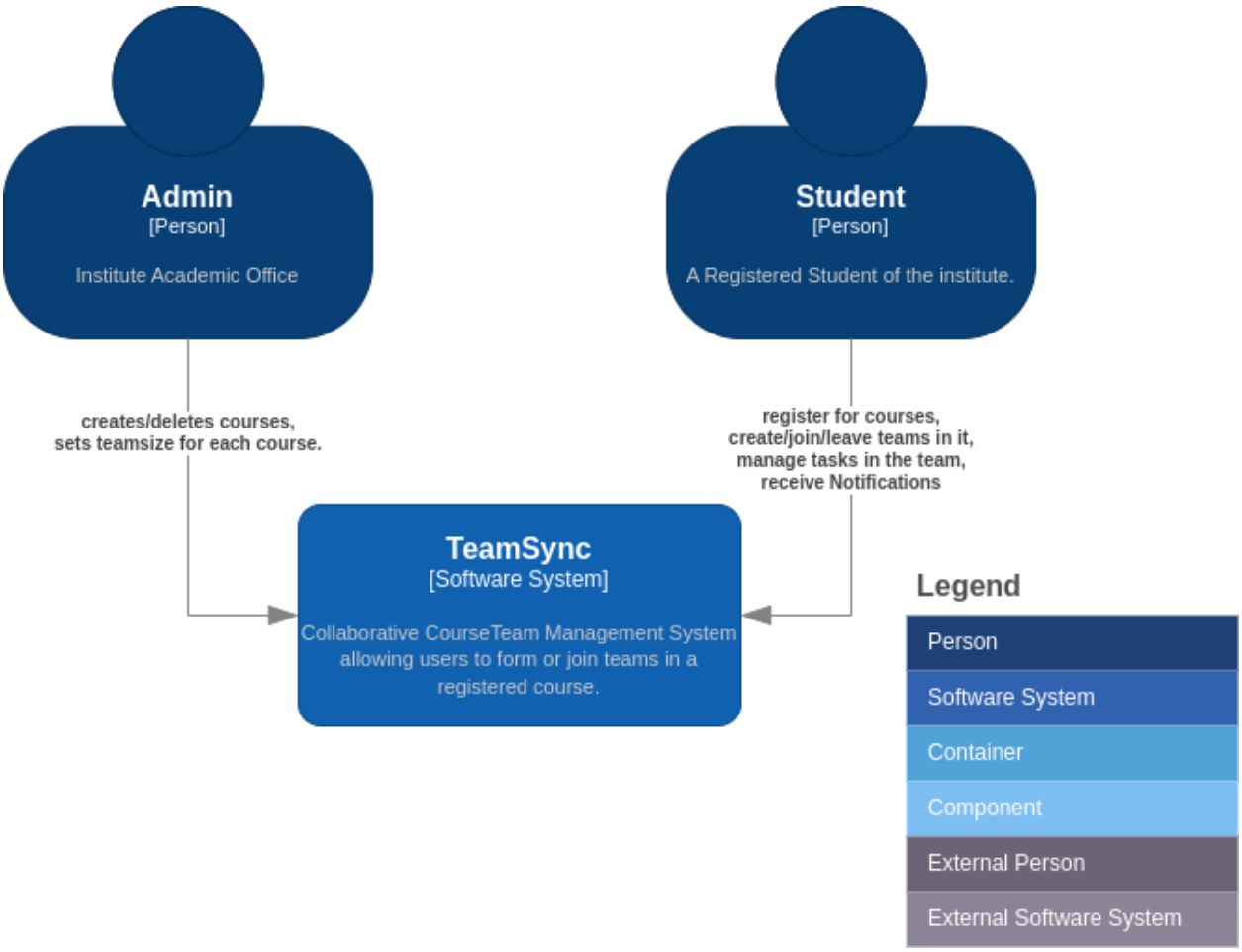
Explanation: The user interface is decoupled from backend services, following a client-server architecture. The frontend interacts with backend APIs, enabling independent development and testing. This separation enhances usability by ensuring a responsive, dynamic, and user-friendly interface while maintaining robust backend functionality.

Addresses:

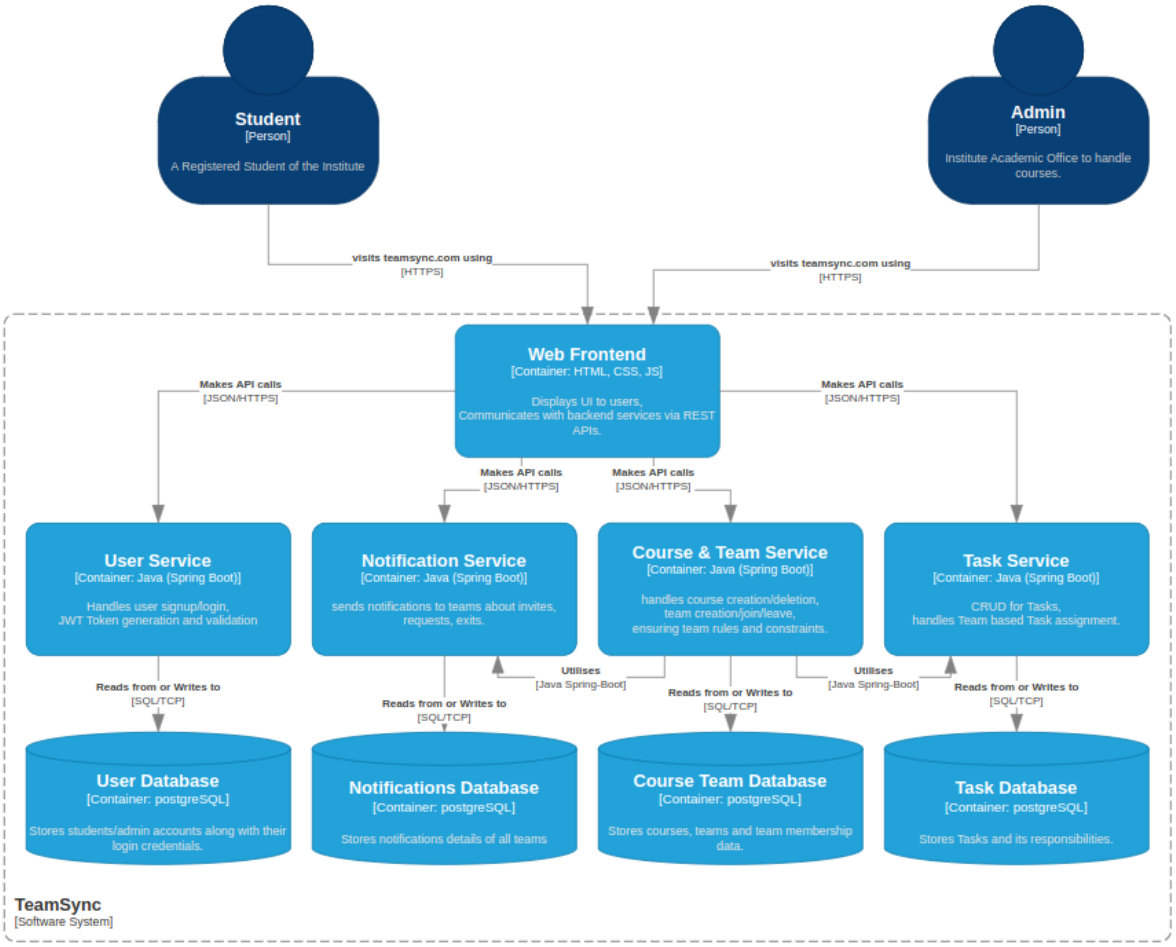
- **NFR6.1:** Ensures an intuitive and responsive user interface.
- Facilitates maintainability and scalability of the frontend and backend independently.

6 Implementation Pattern: C4Model

6.1 C1



6.2 C2


Student
[Person]
A Registered Student of the Institute
Admin
[Person]
Institute Academic Office to handle courses.
visits teamsync.com using
[HTTPS]
visits teamsync.com using
[HTTPS]
Web Frontend
[Container: HTML, CSS, JS]
Displays UI to users,
Communicates with backend services via REST APIs.
Makes API calls
[JSON/HTTPS]
Makes API calls
[JSON/HTTPS]
Makes API calls
[JSON/HTTPS]
Makes API calls
[JSON/HTTPS]
User Service
[Container: Java (Spring Boot)]
Handles user signup/login,
JWT Token generation and validation
Notification Service
[Container: Java (Spring Boot)]
sends notifications to teams about invites, requests, exits.
Course & Team Service
[Container: Java (Spring Boot)]
handles course creation/deletion,
team creation/join/leave,
ensuring team rules and constraints.
Task Service
[Container: Java (Spring Boot)]
CRUD for Tasks,
handles Team based Task assignment.
Reads from or Writes to
[SQL/TCP]
Utilises
[Java Spring-Boot]
Reads from or Writes to
[SQL/TCP]
Reads from or Writes to
[SQL/TCP]
Utilises
[Java Spring-Boot]
Reads from or Writes to
[SQL/TCP]
User Database
[Container: postgreSQL]
Stores students/admin accounts along with their login credentials.
Notifications Database
[Container: postgreSQL]
Stores notifications details of all teams
Course Team Database
[Container: postgreSQL]
Stores courses, teams and team membership data.
Task Database
[Container: postgreSQL]
Stores Tasks and its responsibilities.
TeamSync
[Software System]

6.3 C3

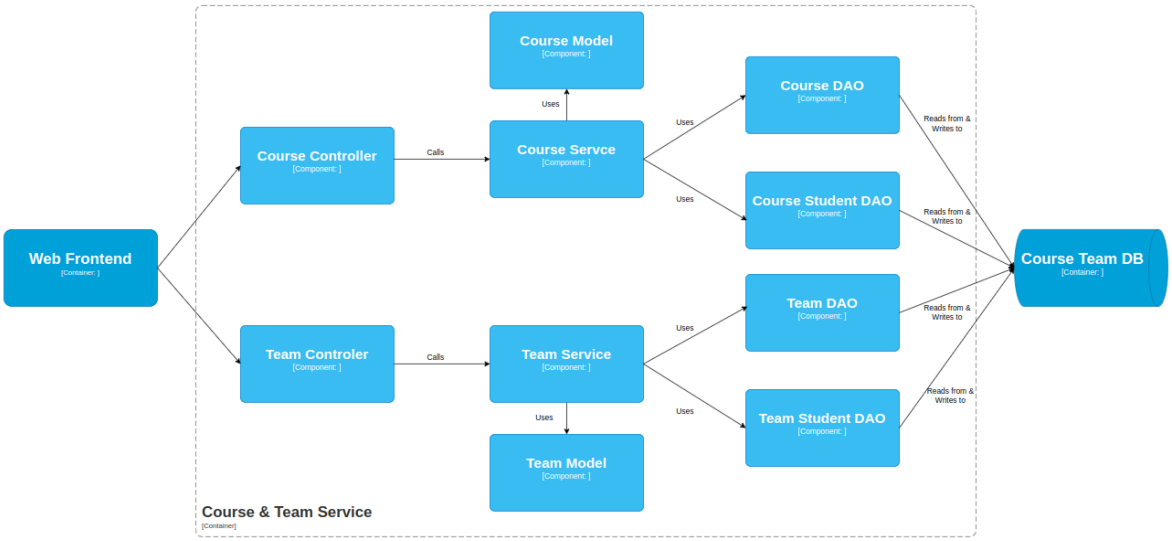


Figure 1: Course and Team Service

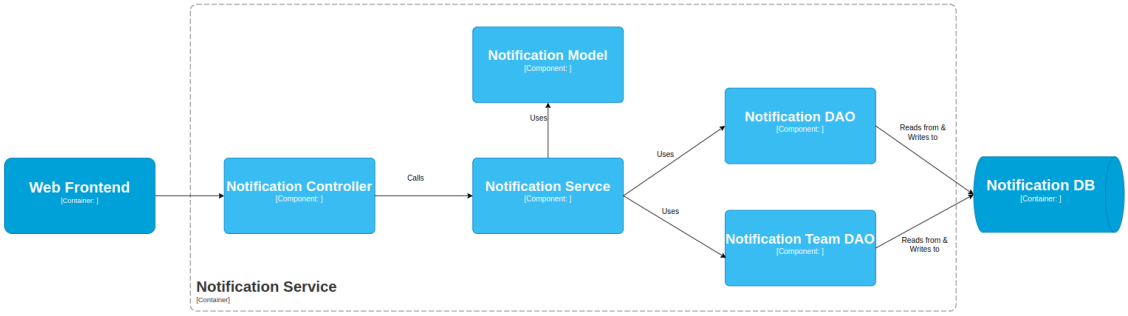


Figure 2: Notification Service

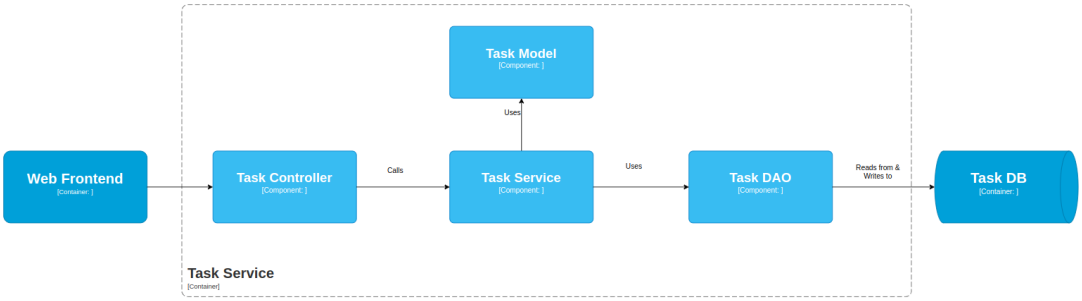


Figure 3: Task Service

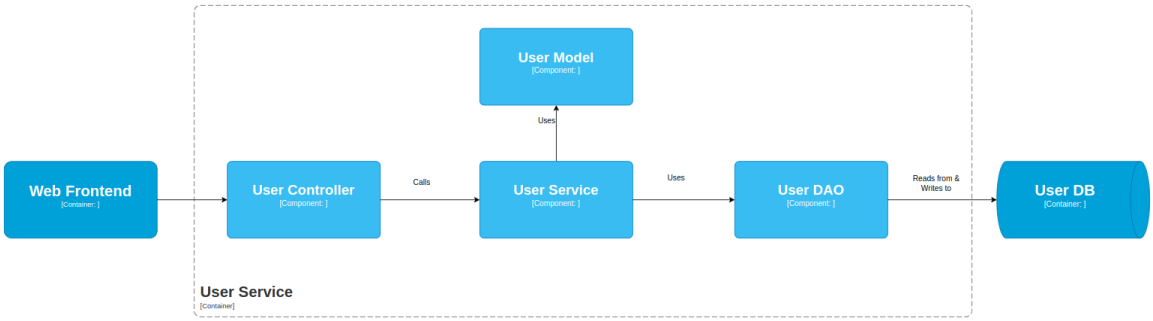


Figure 4: User Service

Task 4: Prototype Implementation and Analysis

We compared the performance of our implemented **Microservices Architecture** against a **Monolithic Architecture**. The tests were conducted using the **Siege tool** with 100 concurrent users over 1 minute. Below are the performance metrics for both implementations.

Metric	Microservices	Monolithic
Transactions	7365	7262
Availability (%)	100.00	100.00
Elapsed Time (s)	59.50	59.88
Data Transferred (MB)	21.46	21.15
Response Time (s)	0.80	0.82
Transaction Rate (per sec)	123.78	121.28
Throughput (MB/s)	0.36	0.35
Concurrency	98.99	99.22
Successful Transactions	5504	5431
Failed Transactions	0	0
Longest Transaction (s)	2.63	2.70
Shortest Transaction (s)	0.00	0.00

6.4 Quantification of Non-Functional Requirements

1. Response Time

Microservices achieved a lower average response time of **0.80 seconds**, compared to **0.82 seconds** for the **Monolithic** architecture. The reduced response time is due to the independent handling of requests in microservices, allowing for better parallelism and concurrency management.

2. Throughput

The throughput for **Microservices** was **0.36 MB/s**, slightly higher than **0.35 MB/s** for the **Monolithic** architecture. This indicates that microservices can handle a marginally larger volume of data transfer in the same time period.

6.5 Trade-Offs and Discussion

Advantages of Microservices:

- **Scalability:** Microservices enable horizontal scaling, allowing specific services to scale independently based on demand.
- **Better Performance:** As evident from the results, microservices demonstrated marginally better performance metrics (response time and throughput).

Disadvantages of Microservices:

- **Increased Complexity:** Managing inter-service communication and distributed deployments adds operational complexity.
- **Latency in Communication:** Though minimal, inter-service communication introduces slight delays compared to intra-process communication in monolithic systems.

Advantages of Monolithic Architecture:

- **Simplicity:** A single codebase and deployment make it easier to manage for small teams.
- **Low Overhead:** Internal communication between components is faster since it does not involve network overhead.

Disadvantages of Monolithic Architecture:

- **Scalability Challenges:** Scaling requires replicating the entire application, even if only specific functionalities need more resources.

6.6 Conclusion

The results demonstrate that **Microservices Architecture** offers marginally better performance for response time and throughput compared to the Monolithic Architecture. However, these benefits come at the cost of increased system complexity and management overhead.

---

7 Contributions

- Chanukya SVSK: User and Notification service.
- Santhosh KM: Course and Team service, Task service.
- Viswanath V: Task service and documentation.