Software Design Specifications

CollabNote

Version: 0.1.0

Project Team	Muhammad Touseef 22k-4328 Huzaifa Ayyaz 22k-4583
Submission Date	May 8, 2025

[Instructions]

- *No section of template should be deleted. You can write ‘Not applicable’ if a section is not applicable to your project. But all sections must exist in the final document.*
- *All comments/examples mentioned in square brackets ([]) are in the template for explanation purposes and must be replaced / removed in final document.*
- *This’ Instruction’ section should also be removed in final document.*
- *MS-Word Reviewing feature must be used to get the document reviewed by PMs or supervisors.*

Document Information

Category	Information
Customer	FAST-NU
Project	<Project Title>
Document	Software Design Specification
Document Version	1.0
Status	Draft
Author(s)	<Names of all the authors of this document>
Approver(s)	
Issue Date	
Document Location	

Category	Information
Distribution	Advisor Project Coordinator’s Office (through Advisor)

Definition of Terms, Acronyms and Abbreviations

[This section should provide the definitions of all terms, acronyms, and abbreviations required to interpret the terms used in the document properly.]

Term	Description
ASP	Active Server Pages
DD	Design Specification

Table of Contents

1 Introduction

1.1 Purpose of Document

The **Software Design Specification (SDS)** document serves as a comprehensive blueprint for the system's architecture and components, outlining how the software will be structured and implemented to meet the specified requirements. Its primary purpose is to provide a clear, detailed representation of the system's design, including data structures, interfaces, components, and workflows, which will guide developers during the implementation phase and ensure consistency across the development lifecycle. This document is intended for a variety of stakeholders including software developers, system architects, project managers, and quality assurance teams, as well as future maintainers who need to understand the system's structure for updates or debugging. For this project, the **Object-Oriented Design (OOD) methodology** will be employed, as it promotes modular, reusable, and maintainable code by organizing the system around objects that represent real-world entities and their interactions.

1.2 Intended Audience

The individuals who are concerned with or expected to use this document include **software developers**, who will reference it to understand the system architecture and guide implementation; **system architects**, who are responsible for reviewing and validating the design decisions; and **project managers**, who use it to track progress and ensure the system aligns with business requirements. Additionally, **quality assurance (QA) engineers** rely on the document to create test plans that align with the design, and **technical support and maintenance teams** refer to it for troubleshooting and future system enhancements. In some cases, **stakeholders or clients** may also review the document to verify that the proposed design meets their expectations and functional requirements.

1.3 Document Convention

[Describe the font and font size that this document will be using]

1.4 Project Overview

The software system is a **project management tool** designed to facilitate collaboration, task tracking, and organization within teams. Its core functionality includes user account management, creation and management of workspaces and projects, assignment and tracking of tasks. Users can belong to multiple workspaces, each containing projects with associated tasks, deadlines, and user assignments. The basic design approach for developing this software will follow an **Object-Oriented Design (OOD) methodology**, which emphasizes encapsulating data and behavior within objects representing real-world entities such as users, tasks, and projects. This approach promotes modularity, scalability, and ease of maintenance, making it well-suited for complex systems with interrelated components.

Scope

The scope of the project is limited to essential project and task management functionalities within defined workspaces. The system will allow users to register and log in, and once authenticated, users can create and manage workspaces. Within each workspace, users with the appropriate role can create, edit, and delete projects and tasks. The system will support role-based access control, with users assigned either as admins (with full control over the workspace) or members (with limited permissions). Additionally, users will be able to join existing workspaces via an invite link, enabling team collaboration.

The system will not include advanced features such as file sharing, real-time chat, calendar integrations, task dependencies, Gantt charts, or third-party integrations. It will also not support advanced permission hierarchies beyond the admin/member roles or provide analytics or reporting features.

2 Design Considerations

This section outlines the key issues and constraints that must be addressed to ensure a well-structured and maintainable system design. These considerations lay the foundation for architectural decisions and guide the development process:

1. **Scalability**: The system must be designed to support multiple users, workspaces, and tasks without performance degradation. Although the initial version will support a basic user base, the design should allow for horizontal or vertical scaling in the future.

2. **Security and Authentication**: User authentication and authorization are crucial. The system must implement secure user registration and login using industry-standard practices such as password hashing and token-based authentication (e.g., JWT). Access to workspace data must be restricted based on user roles.

3. **Role-Based Access Control**: The design must clearly distinguish between the privileges of admins and members within a workspace. Admins can manage all resources, while members may have limited access based on their role.

4. **Data Integrity and Consistency**: The system should ensure data consistency across workspaces, projects, and tasks. This includes preventing unauthorized edits and maintaining referential integrity when deleting or updating entities.

5. **User Experience**: Since the application is intended for team collaboration, it should be intuitive and easy to use. The front end (React) must be responsive and provide a smooth user interface for managing workspaces and tasks.

6. **Invite Mechanism**: A secure and reliable mechanism must be implemented for users to join workspaces via invite links. The system should handle expired or invalid links gracefully and verify permissions before granting access.

7. **Maintainability and Extensibility**: The system should be modular and follow object-oriented principles to allow future enhancements like notifications, activity logs, or integrations with third-party tools.

8. **Technology Stack Compatibility**: The selected technology stack—React for the frontend, Express.js/Node.js for the backend, and SQL for persistent storage—must integrate smoothly and support the intended design patterns and architecture.

2.1 Assumptions and Dependencies

This section highlights additional assumptions and dependencies specific to the system's design phase that were not previously covered in the **Software Requirements Specification (SRS)** document. These issues are critical for making design decisions and ensuring the system's overall functionality:

3 **Database Structure**: It is assumed that the SQL database will have the necessary structure to handle users, workspaces, projects, and tasks efficiently. The design will need to address how relationships between these entities are stored and retrieved, ensuring data integrity and optimal performance.

4 **Session Management**: The system will rely on session management (via JWT or similar) for user authentication. It is assumed that the frontend will handle session persistence through cookies or local storage, and the backend will need to validate these sessions with each request.

5 **Cross-Platform Compatibility**: The system will be designed to function across all modern browsers (Chrome, Firefox, Edge, etc.), but there is an assumption that users will be accessing the platform primarily from desktop devices. The mobile responsiveness of the application will be considered but will not be a primary focus during the initial design phase.

6 **Invite System**: The invite system will depend on the assumption that invite links can be securely generated and that there will be mechanisms to handle link expiration and link reuse. The design will need to consider a user-friendly flow for joining a workspace via these links.

7 **Role Hierarchy**: It is assumed that the system will only support two roles—admin and member—within each workspace. There is no expectation for advanced permission or role management at this time. The design will need to ensure clear role-based access control (RBAC) within the workspace structure

8 **Error Handling and Logging**: It is assumed that there will be basic error handling for all system interactions, and the design will include logging for both user activity and system errors to ensure proper debugging and auditing in the future.

8.1 Risks and Volatile Areas

The system design is subject to several potential sources of change and risks:

1. **Evolving User Requirements**: New features or changes may emerge as users interact with the system. The design will be **modular** and **flexible**, allowing for easy feature additions.

2. **Technological Advancements**: New technologies (e.g., cloud platforms or frameworks) may require updates. The system will be designed to support **scalable architectures** and **service-oriented approaches** to accommodate future tech shifts.

3. **Performance and Scalability**: Growing user base may introduce performance issues. The system will be built for **scalability** with efficient backend design and options for horizontal scaling.

4. **Security and Compliance**: Regulatory changes (e.g., GDPR) or new security threats could impact design. The system will implement **robust security practices** and be flexible for future compliance updates.

5. **Integration with External Systems**: As the system grows, third-party integrations may be required. The design will support **extensibility** with API integrations.

6. **UI/UX Feedback**: Changes may be needed based on user feedback. The system will be **responsive and user-centered**, with room for UI adjustments.

7. **Mobile and Cross-Platform Needs**: Future demand for mobile support may arise. The frontend will be **responsive** to accommodate mobile devices.

Contingency Planning for Changes

9 **Modular Design**: Ensures easy addition of new features without major rewrites.

10 **Version Control and API Flexibility**: Allows backward-compatible updates and proper versioning.

11 **Scalable Infrastructure**: Ensures horizontal scalability with cloud infrastructure and microservices.

12 **CI/CD**: Continuous deployment for smooth updates.

13 **User Feedback Loops**: Regular testing ensures iterative improvements.

14 System Architecture

The system is divided into key subsystems and components to provide the required project management functionality while ensuring scalability, maintainability, and ease of use. Below is a high-level breakdown of how the functionality is partitioned and assigned to various components:

1. **User Management Subsystem**:
    - **Responsibilities**: Handles user authentication (registration and login), role management (admin/member), and session management.
    - **Components**:
        - **Authentication Service**: Manages login, registration, and token-based authentication (e.g., JWT).
        - **User Service**: Manages user data, such as personal information, roles, and workspace memberships.
2. **Workspace Management Subsystem**:
    - **Responsibilities**: Manages the creation, editing, and deletion of workspaces, as well as inviting members to join.
    - **Components**:
        - **Workspace Service**: Handles workspace creation, editing, deletion, and member role assignments.
        - **Invite Service**: Manages the creation of invite links, including validation and expiration.
3. **Project Management Subsystem**:
    - **Responsibilities**: Manages the creation, editing, and deletion of projects within workspaces.
    - **Components**:
        - **Project Service**: Handles CRUD (Create, Read, Update, Delete) operations for projects.
        - **Task Service**: Manages tasks associated with projects, allowing tasks to be created, edited, and deleted.
4. **Database Subsystem**:
    - **Responsibilities**: Provides data persistence for users, workspaces, projects, and tasks.

- **Components**:
    - **User Database**: Stores user credentials, profiles, and roles.
    - **Workspace Database**: Stores workspace details, including members and roles.
    - **Project and Task Database**: Stores project and task data, including relationships between tasks and projects.

5. **Frontend (User Interface)**:
    - **Responsibilities**: Provides an interface for users to interact with the system, including workspace management, task creation, and project tracking.
    - **Components**:
        - **React Application**: The core frontend component for user interaction, built with React to handle user interface updates and state management.
6. **Backend (API)**:
    - **Responsibilities**: Exposes the system's functionality to the frontend through a REST or GraphQL API. It handles requests related to workspaces, projects, tasks, and user data.
    - **Components**:
        - **Node.js/Express Server**: Provides the API endpoints, routing, and business logic.

Interaction Between Components:

- The **Frontend (React)** communicates with the **Backend API** to perform CRUD operations on workspaces, projects, and tasks.
- The **Backend (Node.js/Express)** interacts with the **Database** to persist and retrieve data as needed.
- **Authentication** and **Role-based Access Control** are enforced by the backend to ensure users can only access resources they are authorized to view or modify.
- The **Invite Service** ensures users can join workspaces via invite links, integrating seamlessly with the **User Management Subsystem**.

This partitioned approach ensures that each component is responsible for a specific set of functionalities, allowing for clear separation of concerns and scalability as the system grows.

15 Design Strategy

The overall system architecture is shaped by several key design strategies that impact its organization, scalability, maintainability, and future extensibility. Below are the core strategies and decisions employed in the design:

1. Modular and Component-Based Architecture

- ***Reasoning****: The system is designed using a* ***modular architecture*** *where each major functionality (e.g., user management, workspace management, task management) is encapsulated in its own subsystem or component. This aligns with the principle of* ***separation of concerns*** *and ensures that each component can be developed, tested, and maintained independently.*

- ***Trade-offs****: While a modular approach improves maintainability and scalability, it also introduces complexity in terms of communication between components. API calls between services must be optimized to avoid performance bottlenecks.*

- ***Future Extension****: This modular structure makes it easier to extend the system by adding new features, such as additional roles or external integrations, without affecting existing components.*

2. Object-Oriented Design (OOD) and Domain Modeling

- ***Reasoning****: The system employs* ***object-oriented principles*** *to model real-world entities like users, tasks, projects, and workspaces. This approach provides* ***encapsulation****,* ***inheritance****, and* ***polymorphism****, allowing easy modification and extension of the system.*

- ***Trade-offs****: Object-oriented design promotes reusability and scalability but may introduce overhead when dealing with simple, flat structures. Some operations may require multiple method calls or data transformations due to the abstraction layers.*

- ***System Reuse****: The use of classes and objects facilitates the reuse of code. For example, the task model can be extended to include additional attributes, while the user model can be reused in different contexts or integrated into other systems (e.g., team management tools).*

3. Role-Based Access Control (RBAC)

- ***Reasoning****: The system adopts a* ***role-based access control*** *strategy to manage permissions across workspaces. Users are assigned roles (admin/member) that define their level of access to workspace data and actions. This strategy ensures that only authorized users can create, modify, or delete resources.*

- ***Trade-offs****: While RBAC simplifies user management and increases security by limiting access, it can be rigid. If more granular roles or permissions are needed later, significant changes may be required to accommodate them.*

- ***Future Extension****: The RBAC system can be extended by introducing additional roles or more granular permissions (e.g., read-only access, task assigner), allowing for future flexibility in access control.*

4. Data Management and Persistence

- ***Reasoning***: *The system uses a* ***relational SQL database*** *(e.g., PostgreSQL or MySQL) to store user, workspace, project, and task data. Relational databases were chosen due to their strong support for* ***data integrity***, ***consistency***, *and* ***complex queries***. *The use of foreign keys ensures referential integrity between workspaces, projects, and tasks.*

- ***Trade-offs***: *SQL databases are suitable for structured data with complex relationships but may struggle with high-volume transactions or unstructured data. For high scalability or future feature growth, NoSQL solutions (e.g., MongoDB) may be considered if the need arises.*

- ***Data Distribution***: *The database design also considers future scalability needs, allowing for partitioning or sharding if the database grows too large.*

5. Scalability and System Extension

- ***Reasoning***: *The design supports* ***horizontal scalability*** *by using cloud services for backend hosting and employing* ***stateless services*** *where possible. This ensures the system can handle increased user load and data volume by simply adding more resources (e.g., more server instances or databases).*

- ***Trade-offs***: *Horizontal scaling introduces complexity in terms of load balancing, session management, and ensuring data consistency across instances. However, this is balanced by the ability to grow as demand increases, avoiding the limitations of vertical scaling.*

- ***Future Extension***: *The system is designed with future enhancements in mind, such as the addition of new user roles, third-party integrations, or advanced features like Gantt charts, task dependencies, or notifications.*

6. Concurrency and Synchronization

- ***Reasoning***: *The system needs to handle concurrent users who may be modifying tasks, projects, or workspaces simultaneously.* ***Concurrency management*** *is critical to ensure data consistency and avoid conflicts. This is managed through* ***optimistic concurrency control***, *where changes to tasks or projects are validated before committing them to the database.*

- ***Trade-offs***: *Optimistic concurrency control reduces locking overhead but may lead to conflicts when multiple users attempt to update the same resource concurrently. This could necessitate additional conflict resolution logic, such as versioning or rollback mechanisms.*

- ***Synchronization***: *To maintain data consistency, the system will implement synchronization mechanisms like transaction management, ensuring that tasks and project modifications are atomic and consistent.*

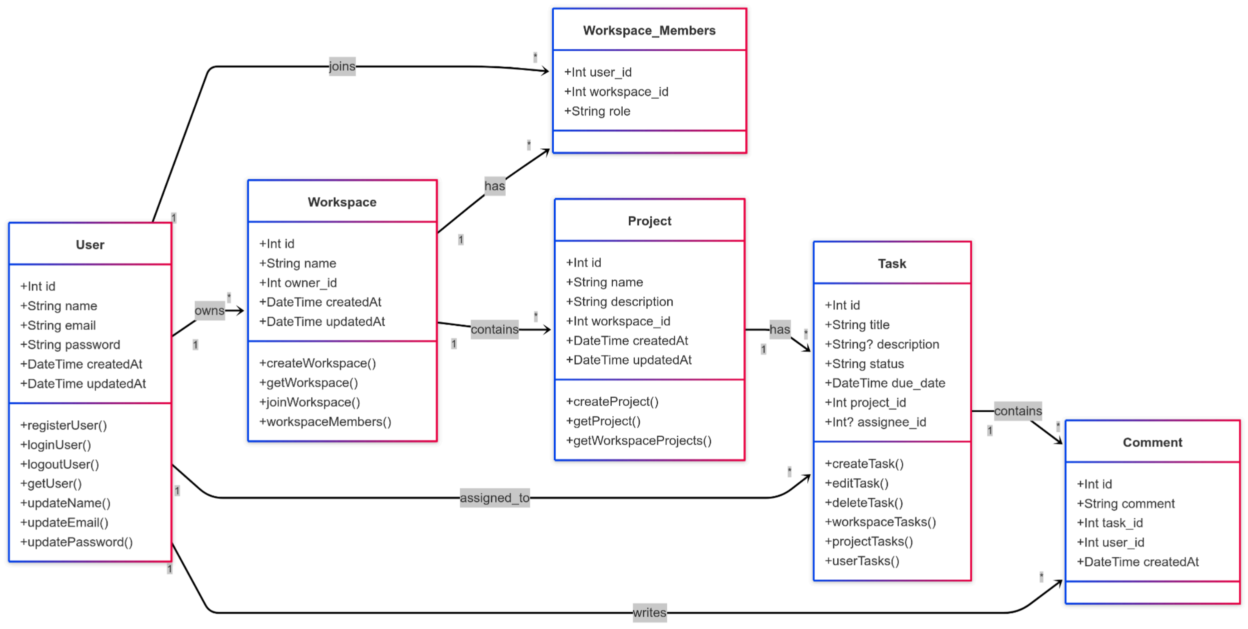
7. User Interface Design

- ***Reasoning***: *The user interface will follow* ***responsive design*** *principles, ensuring it works seamlessly across desktop and mobile devices.* ***React*** *is used for dynamic, interactive UI components, providing a smooth and efficient user experience.*

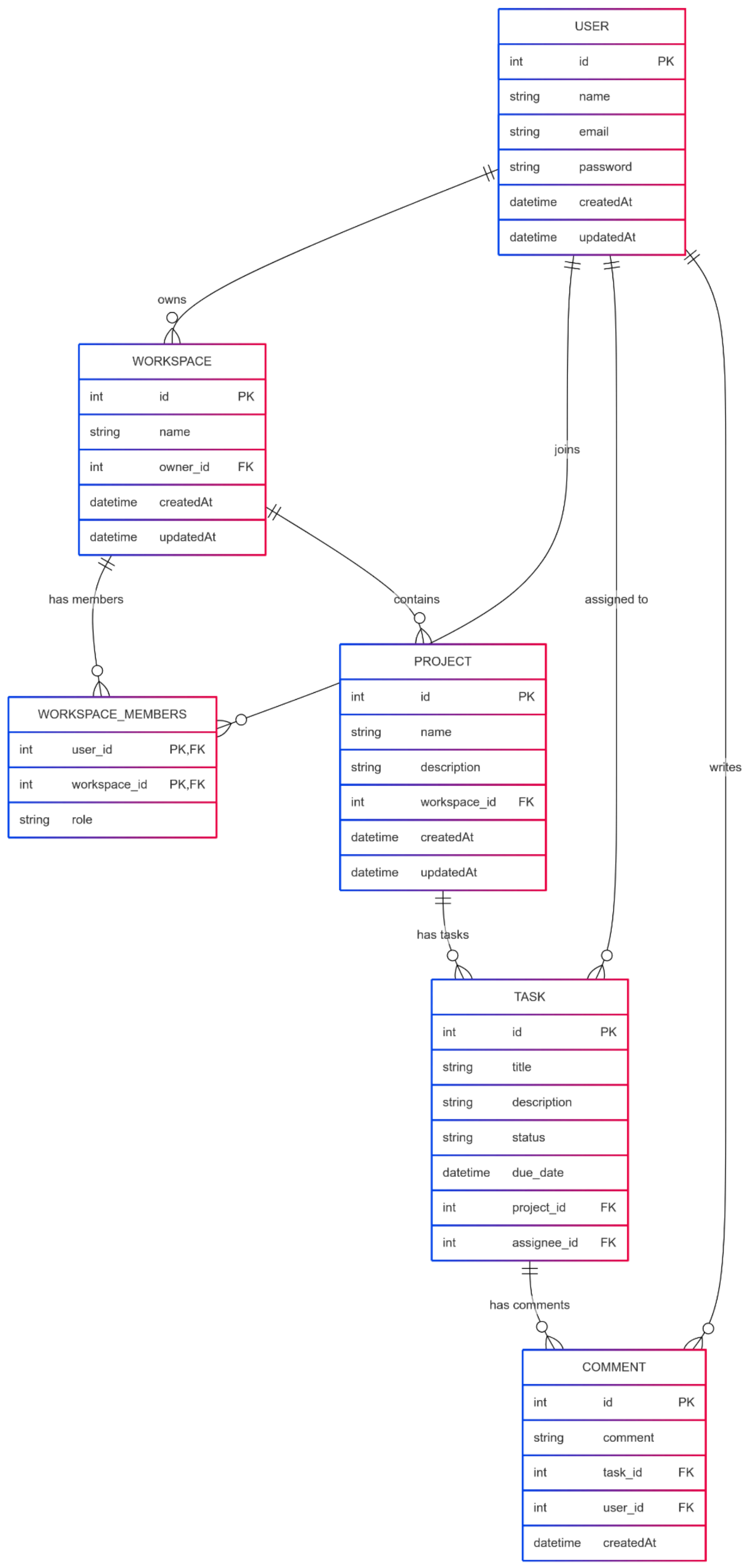
- ***Trade-offs***: *While React offers flexibility and performance, it requires careful management of state and components to ensure that the UI remains responsive as data changes. This may involve additional overhead in terms of managing component lifecycle and state consistency.*

- ***User Interface Paradigms**: The design will prioritize a **clean, intuitive UI**, with an emphasis on usability and minimalism, ensuring that users can easily navigate through workspaces, projects, and tasks. This will be achieved through modular, component-based UI elements.*

16 Detailed System Design



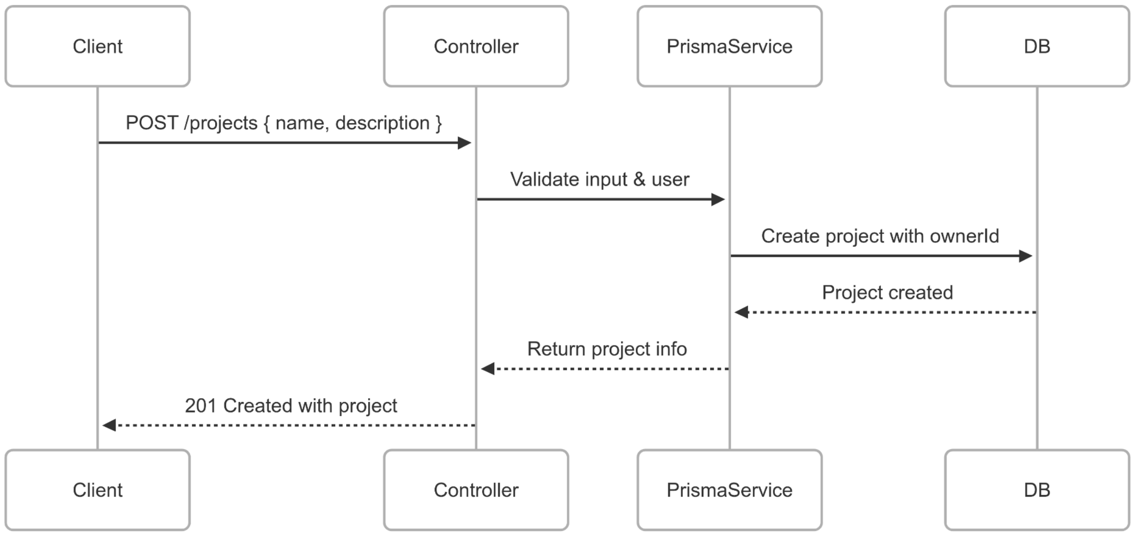
16.1 Database Design


USER
int id PK
string name
string email
string password
datetime createdAt
datetime updatedAt
owns
WORKSPACE
int id PK
string name
int owner_id FK
datetime createdAt
datetime updatedAt
joins
assigned to
writes
has members
contains
PROJECT
int id PK
string name
string description
int workspace_id FK
datetime createdAt
datetime updatedAt
WORKSPACE_MEMBERS
int user_id PK,FK
int workspace_id PK,FK
string role
has tasks
TASK
int id PK
string title
string description
string status
datetime due_date
int project_id FK
int assignee_id FK
has comments
COMMENT
int id PK
string comment
int task_id FK
int user_id FK
datetime createdAt

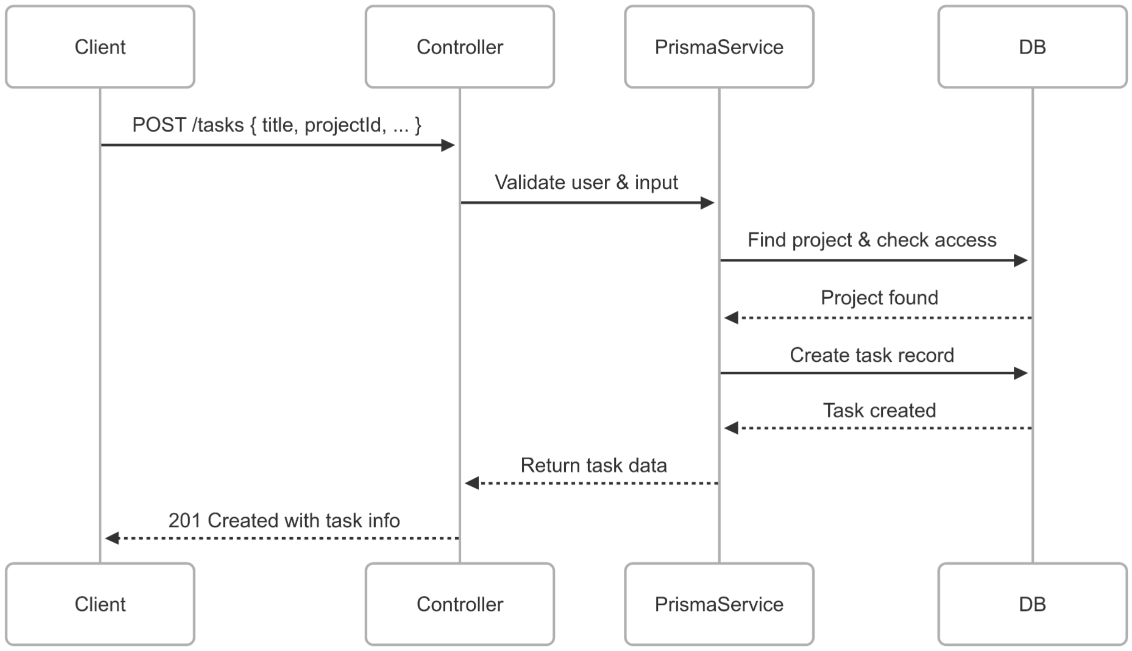
16.2 Application Design

16.2.1 Sequence Diagram

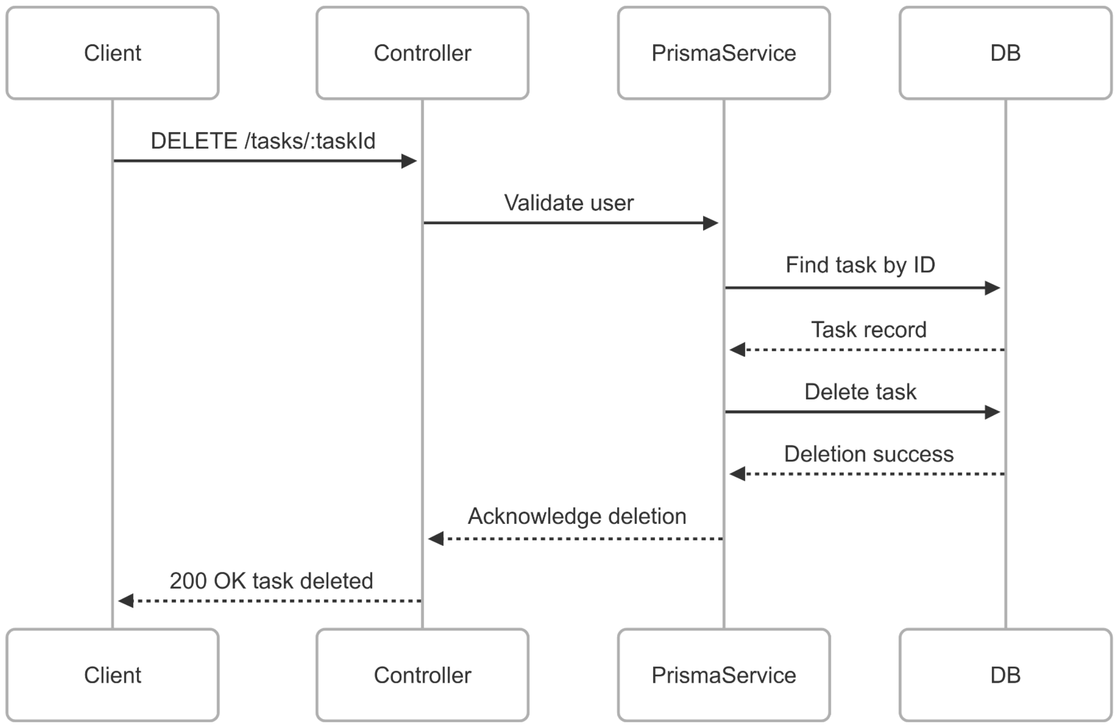
16.2.1.1<Create Project>



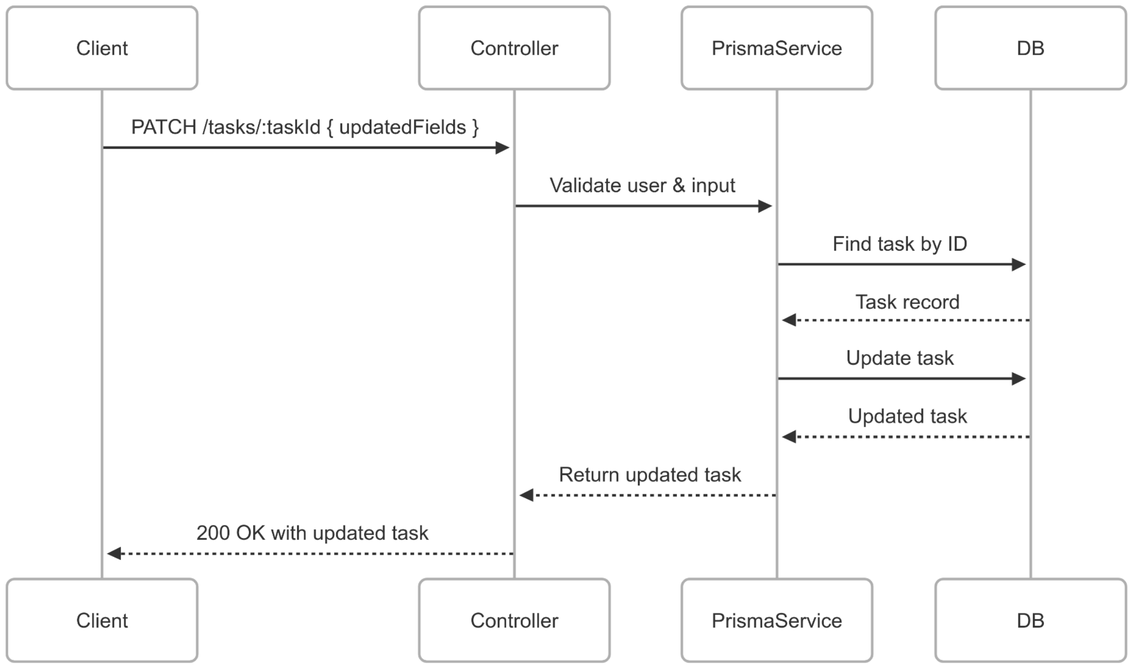
16.2.1.2<Create Task>



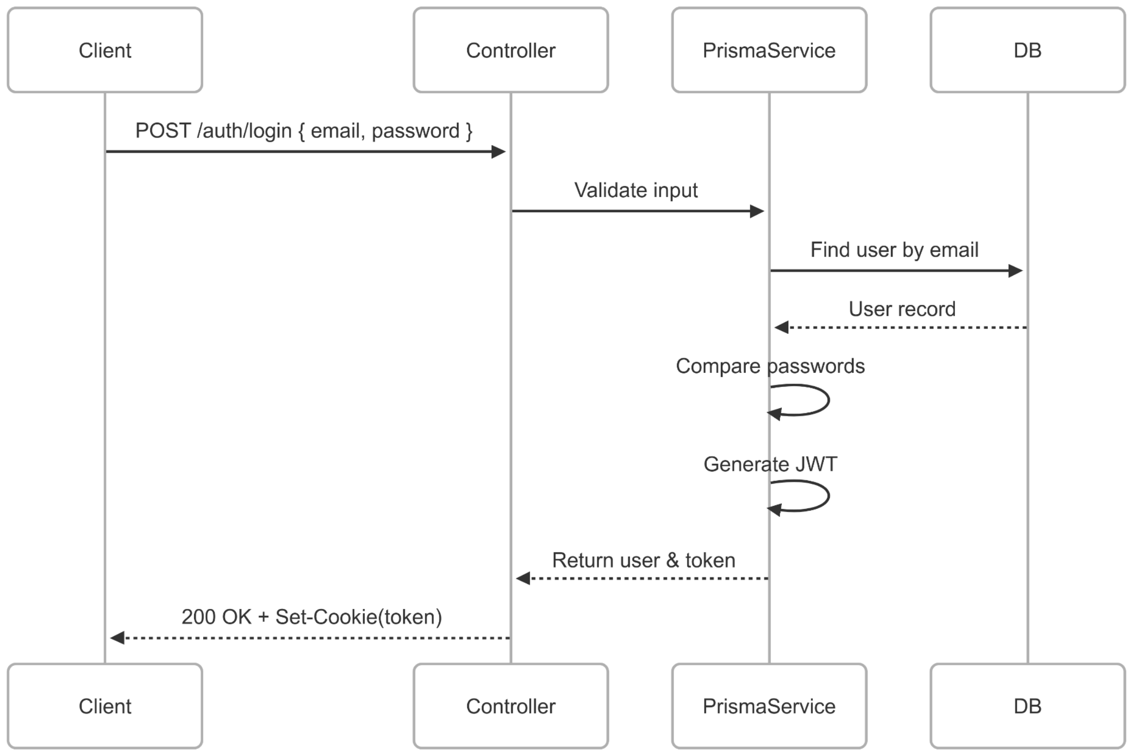
16.2.1.3 <Delete Task>



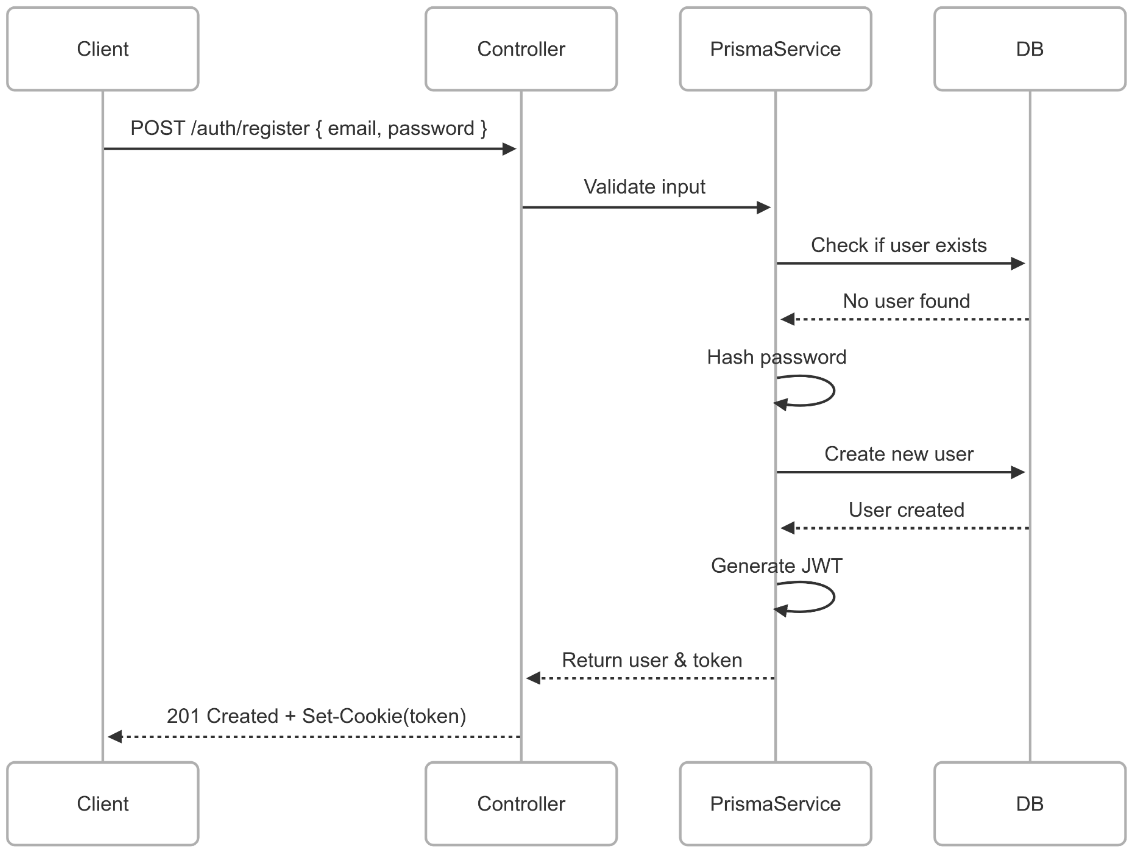
16.2.1.4<Edit Task>



16.2.1.5<Login User>

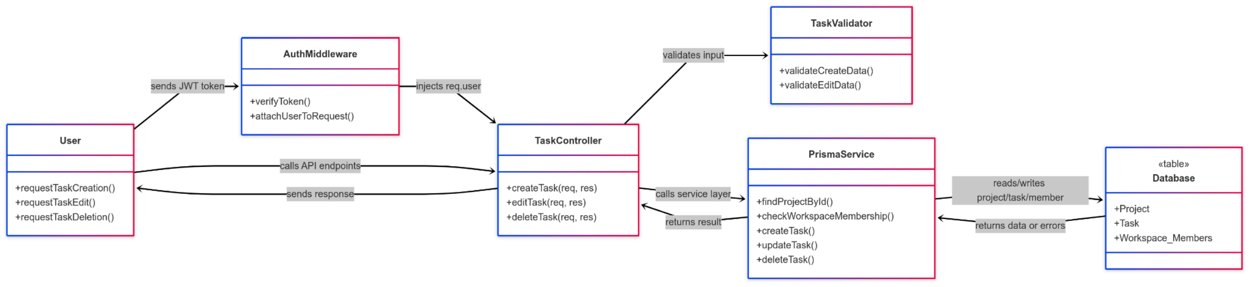

Client
Controller
PrismaService
DB
POST /auth/login { email, password }
Validate input
Find user by email
User record
Compare passwords
Generate JWT
Return user & token
200 OK + Set-Cookie(token)
Client
Controller
PrismaService
DB

16.2.1.6<Register User>

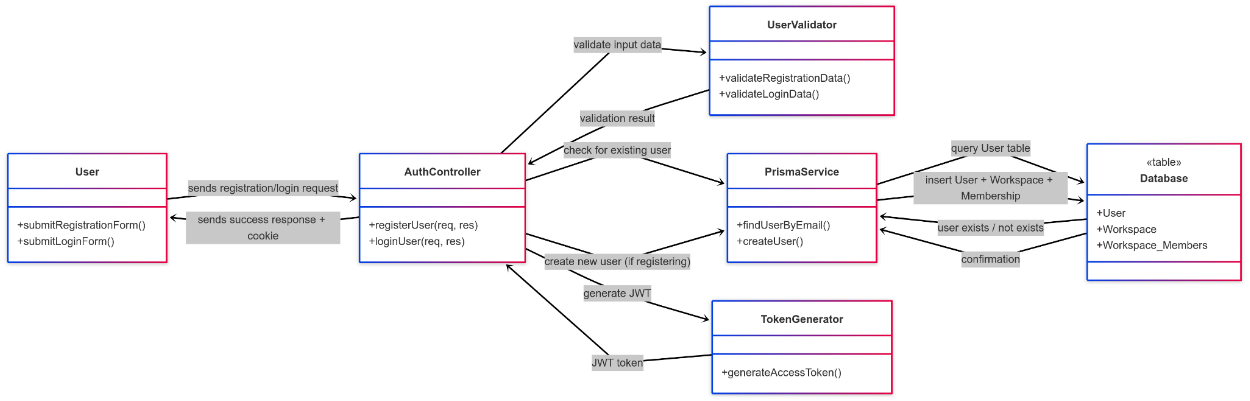

Client
Controller
PrismaService
DB
POST /auth/register { email, password }
Validate input
Check if user exists
No user found
Hash password
Create new user
User created
Generate JWT
Return user & token
201 Created + Set-Cookie(token)
Client
Controller
PrismaService
DB

16.2.2 State Diagram

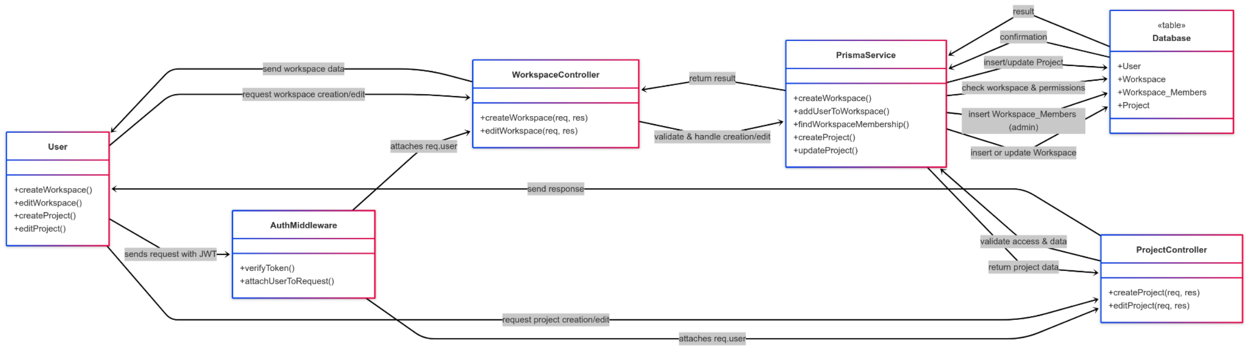
16.2.2.1<Task workflow>



16.2.2.2<user workflow>

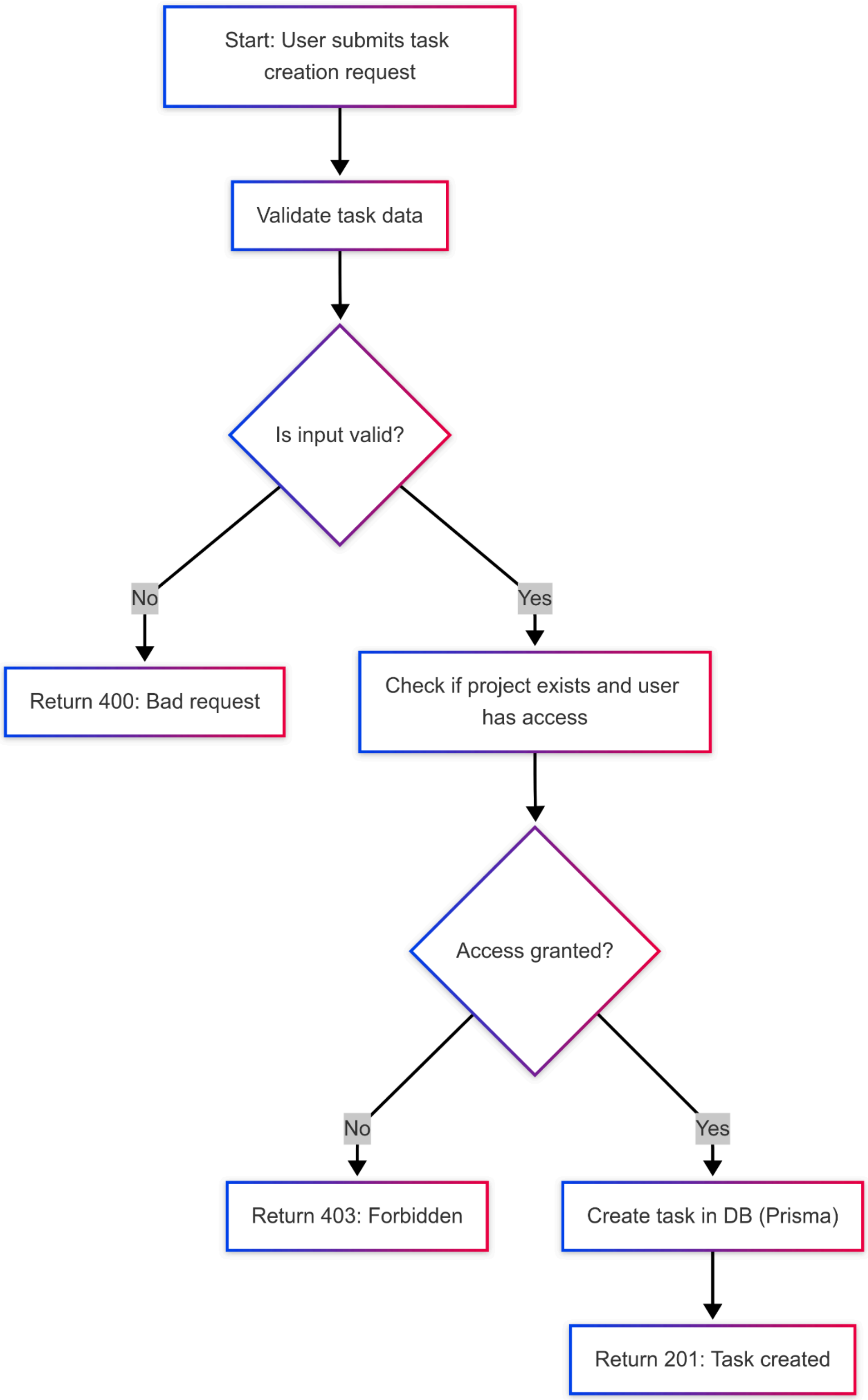


16.2.2.3<Project and workspace workflow>

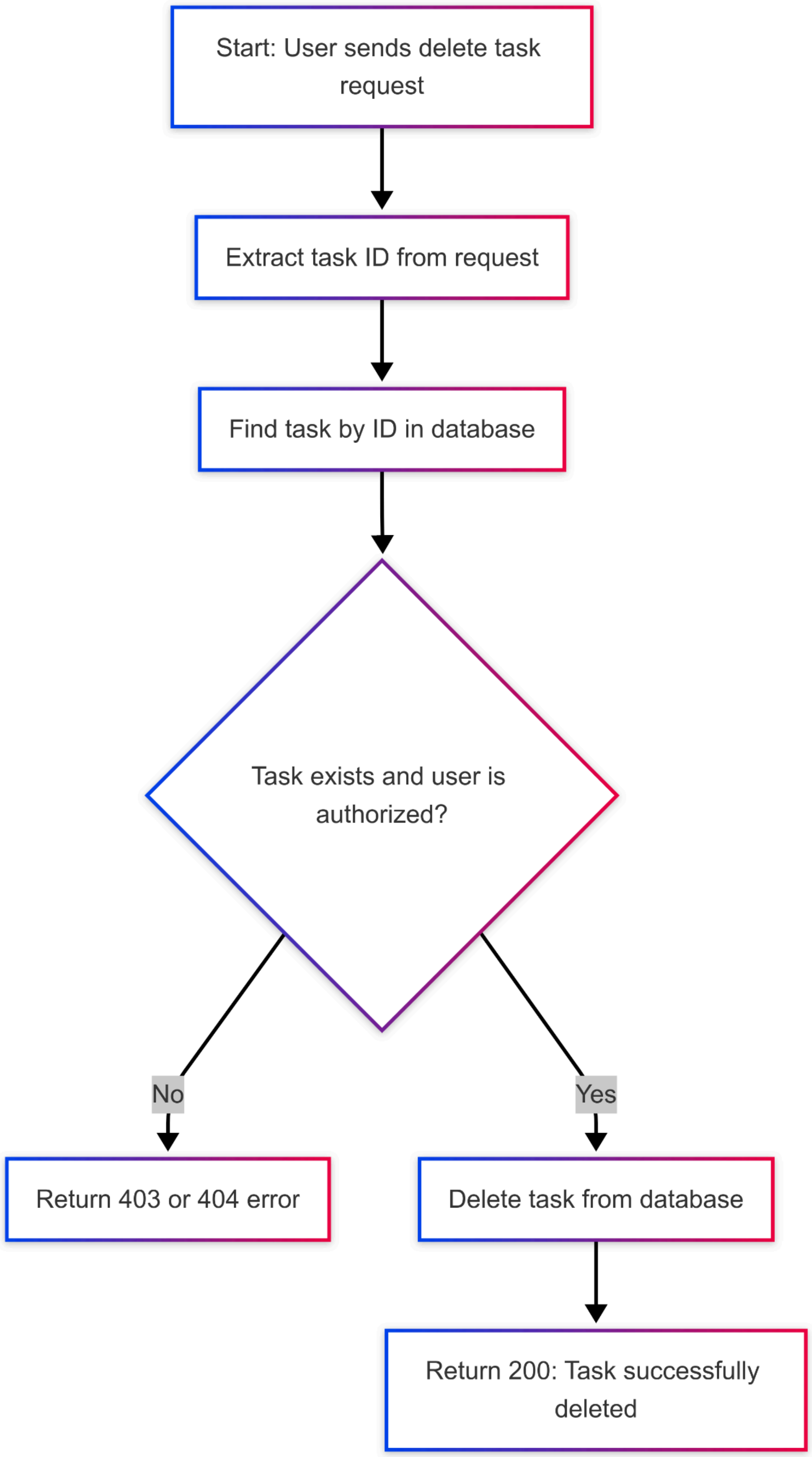


16.2.4 Activity Diagram

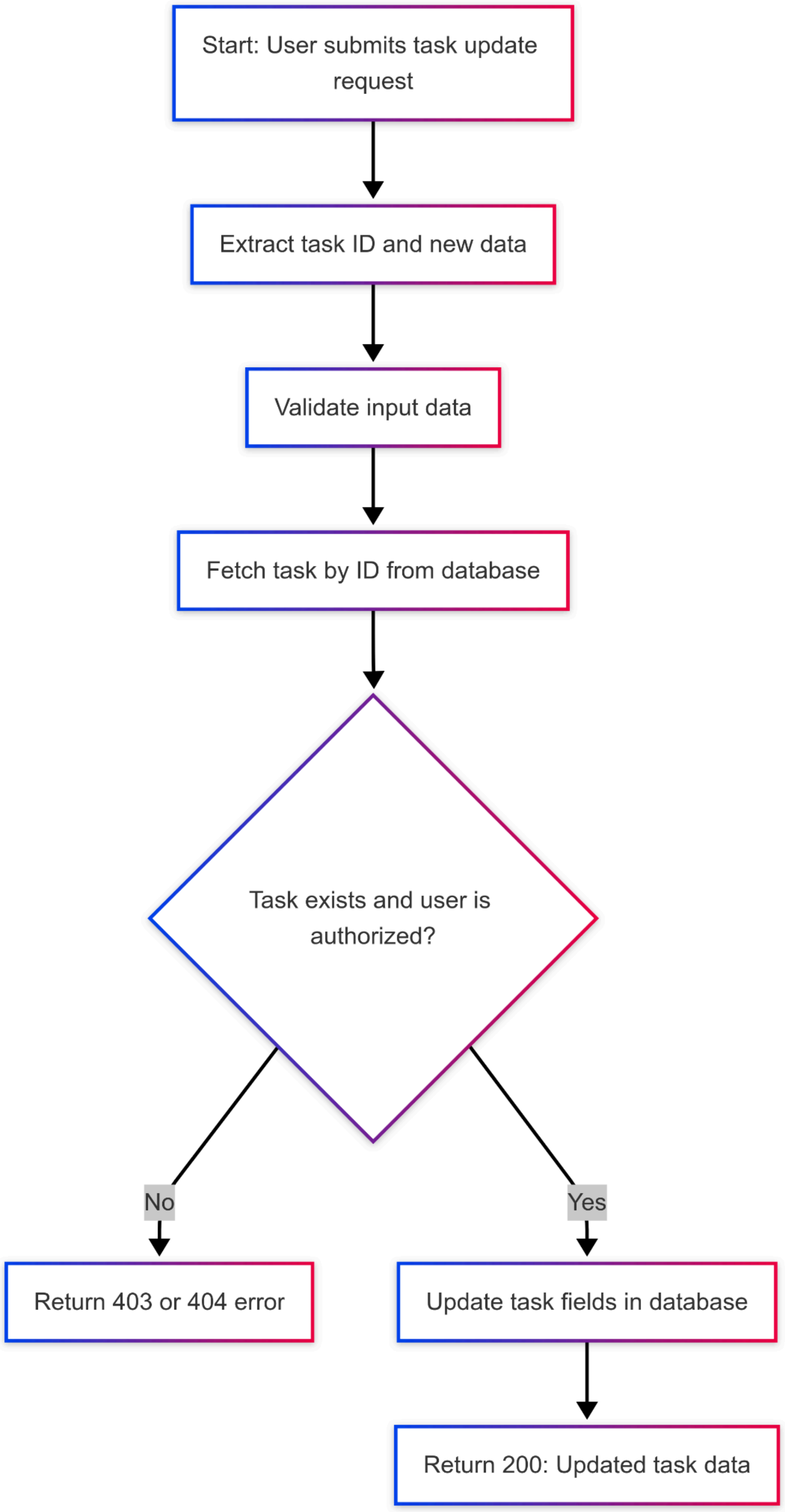
16.2.4.1<Create Task>


Start: User submits task creation request
Validate task data
Is input valid?
No
Yes
Return 400: Bad request
Check if project exists and user has access
Access granted?
No
Yes
Return 403: Forbidden
Create task in DB (Prisma)
Return 201: Task created

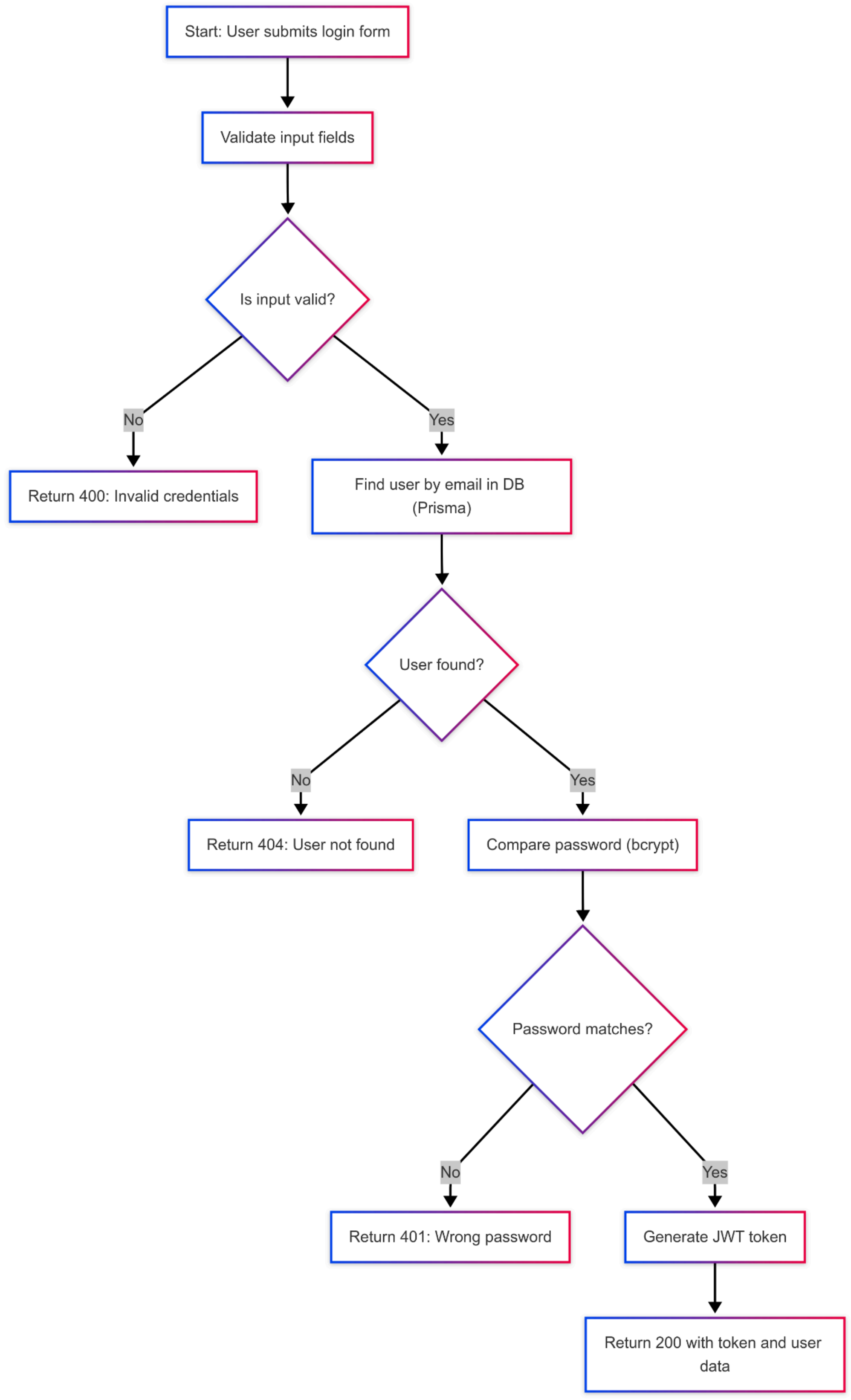
16.2.4.1<Delete Task>


Start: User sends delete task request
Extract task ID from request
Find task by ID in database
Task exists and user is authorized?
No
Yes
Return 403 or 404 error
Delete task from database
Return 200: Task successfully deleted

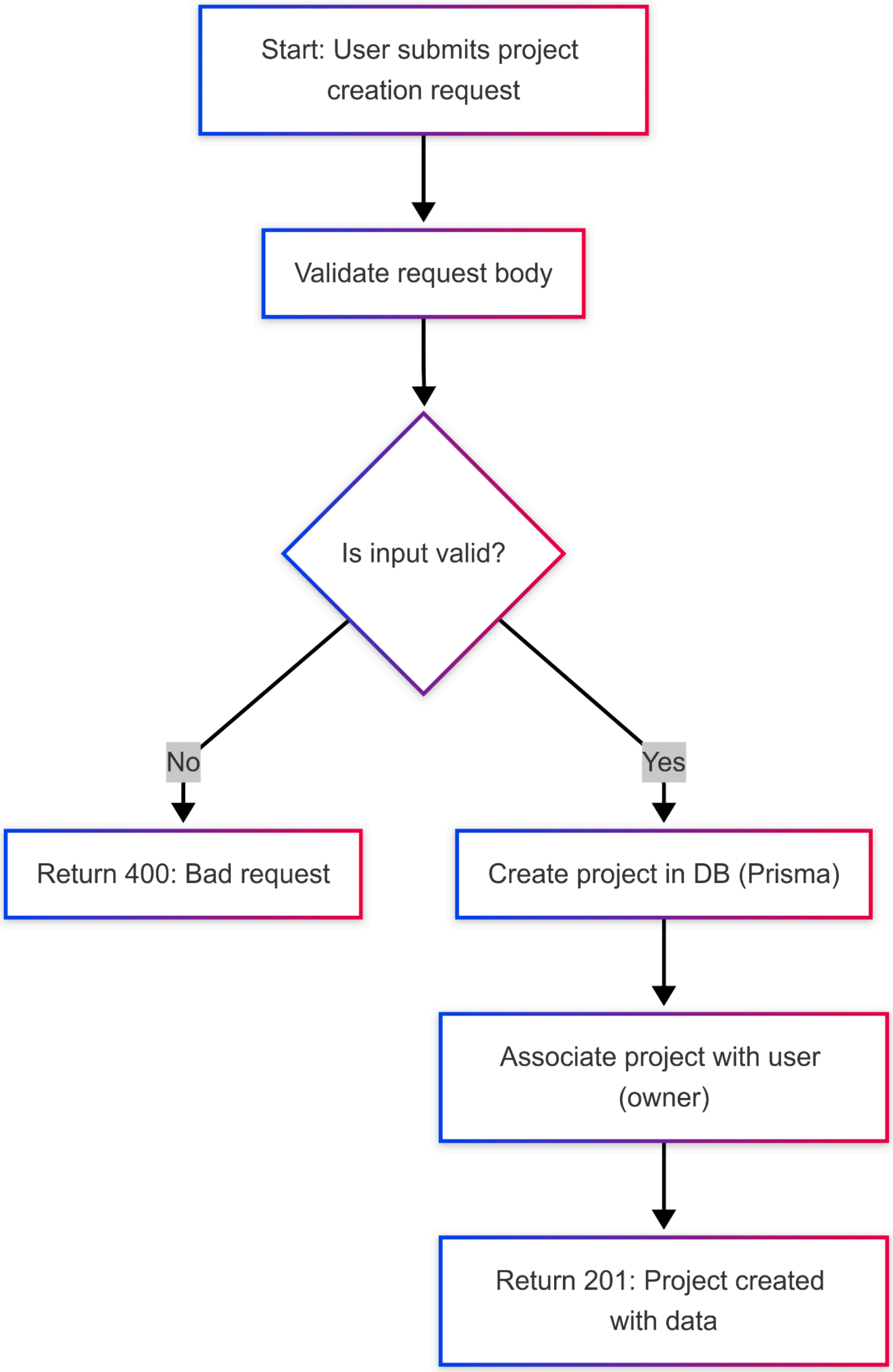
16.2.4.1<Edit Task>


Start: User submits task update request
Extract task ID and new data
Validate input data
Fetch task by ID from database
Task exists and user is authorized?
No
Yes
Return 403 or 404 error
Update task fields in database
Return 200: Updated task data

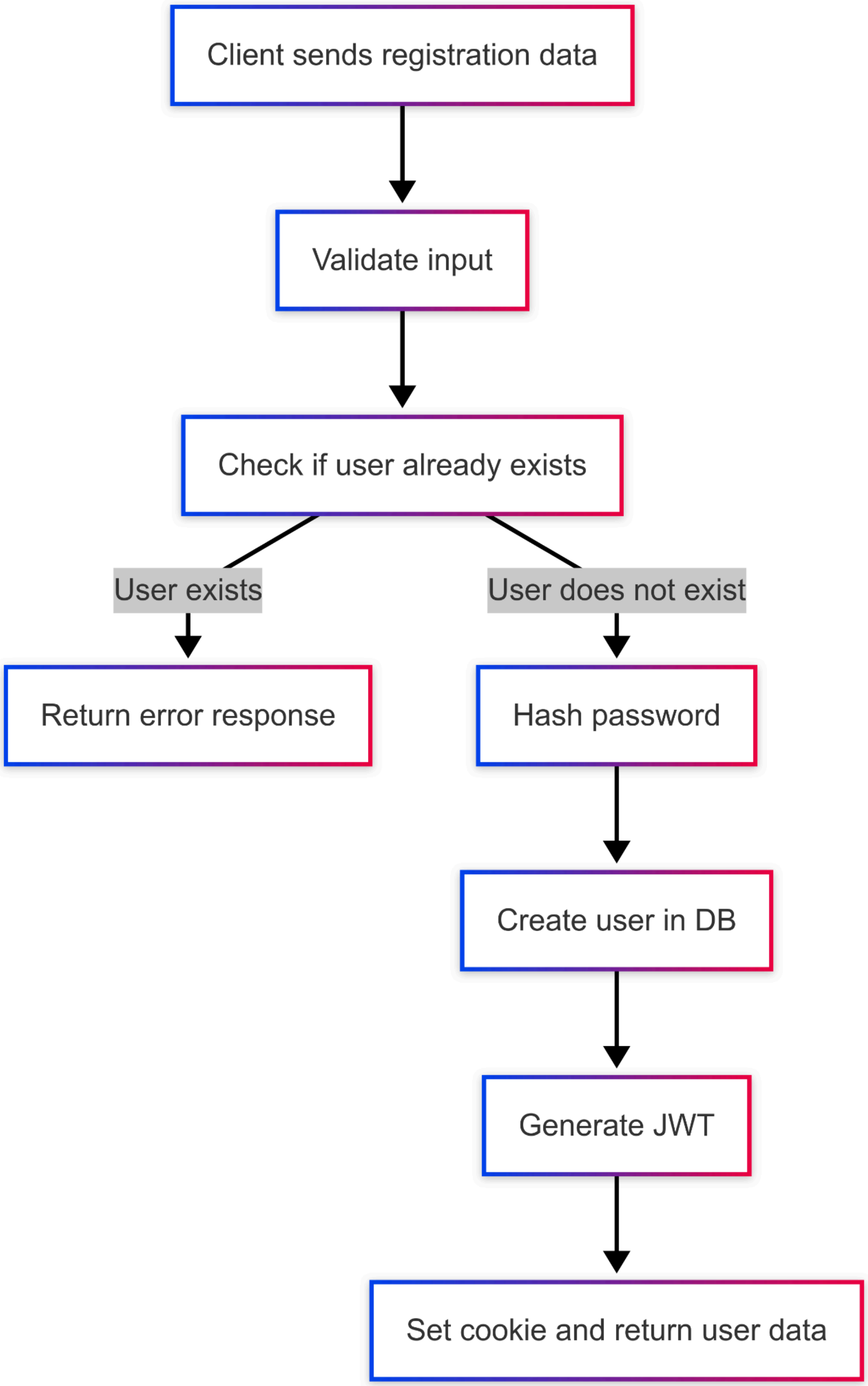
16.2.4.1<Login>


Start: User submits login form
Validate input fields
Is input valid?
No
Yes
Return 400: Invalid credentials
Find user by email in DB (Prisma)
User found?
No
Yes
Return 404: User not found
Compare password (bcrypt)
Password matches?
No
Yes
Return 401: Wrong password
Generate JWT token
Return 200 with token and user data

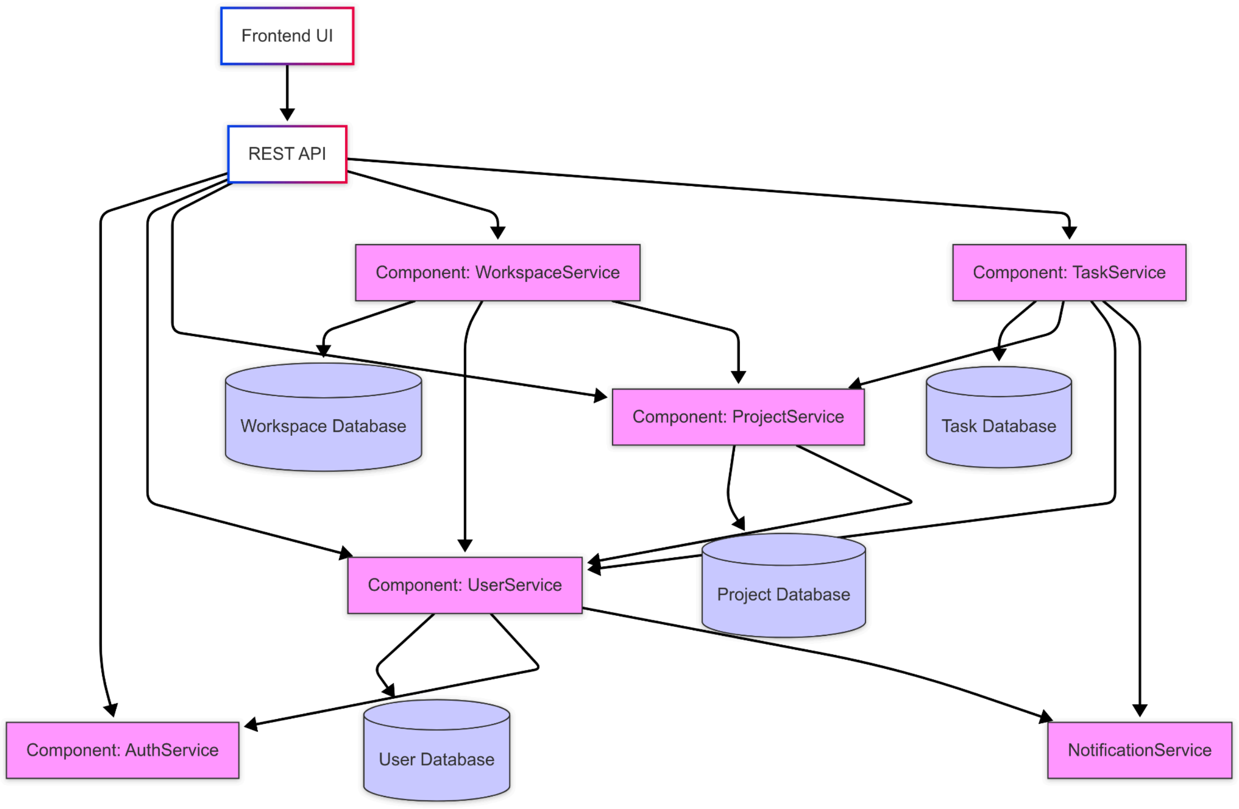
16.2.4.1<Project creation>


Start: User submits project creation request
Validate request body
Is input valid?
No
Yes
Return 400: Bad request
Create project in DB (Prisma)
Associate project with user (owner)
Return 201: Project created with data

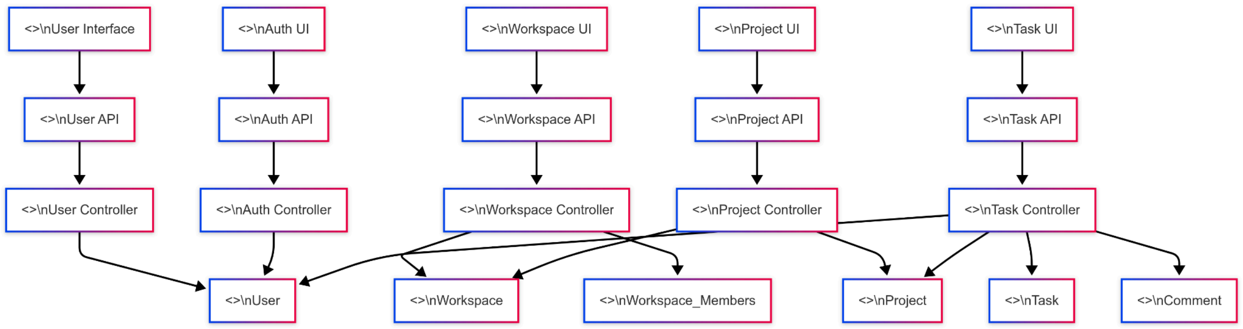
16.2.4.1<Register>



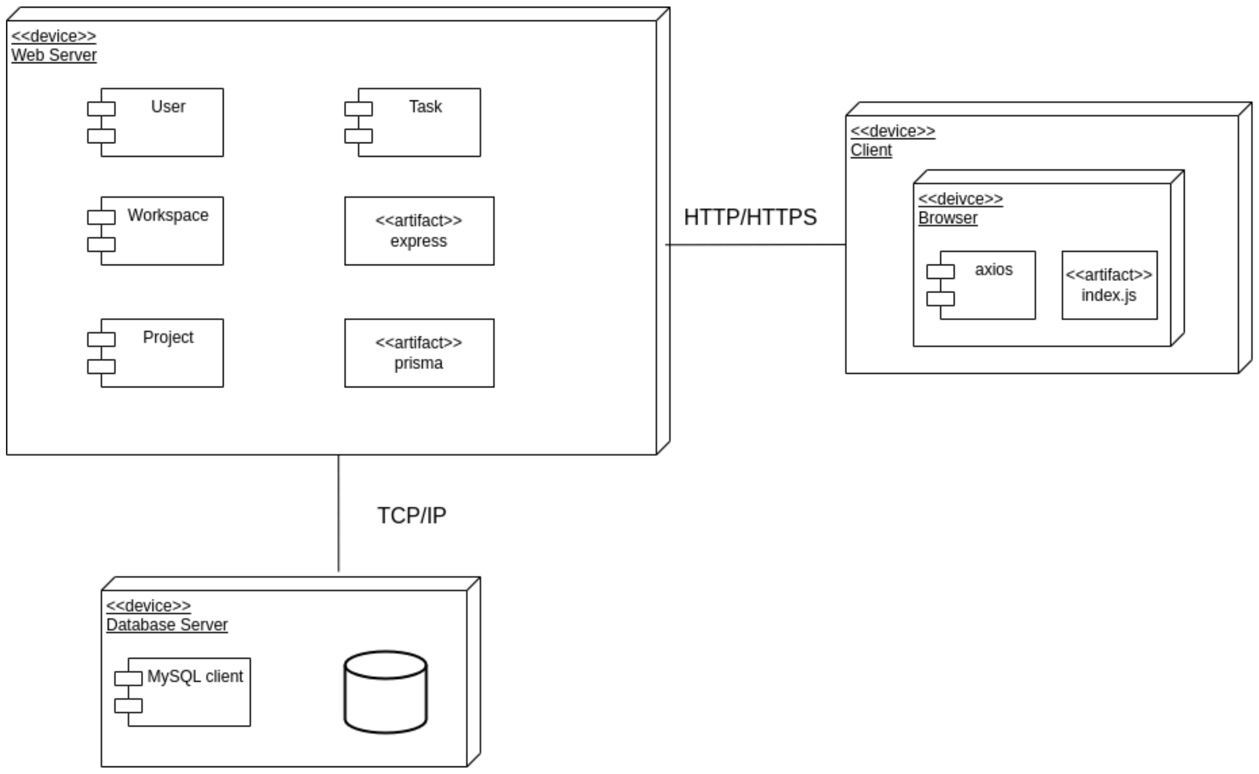
Component Diagram



ECB Diagram



Deployment Diagram


<<device>>
Web Server
User
Task
Workspace
<<artifact>> express
Project
<<artifact>> prisma
HTTP/HTTPS
<<device>>
Client
<<deivce>>
Browser
axios
<<artifact>> index.js
TCP/IP
<<device>>
Database Server
MySQL client