CS319 Course Project

Deliverable 4

Software Design Description

Team Members:

Ali Şen / 22203368

Barış Peksak / 22201659

Berkay Şimşek / 22303338

Gökay Nuray / 22302913

Muhammet Furkan Demir / 22302644

Contents

Design Goals, Connectors, and Architectural Style

Our system was built around five primary design goals: security, scalability, maintainability, interoperability, and data integrity. For security, we enforced strict role-based access control distinguishing Researchers from Participants and selected JWT-based stateless authentication instead of server-side sessions. Although JWTs complicate immediate revocation and shift token responsibility to the client, they remove server-side session state and allow the backend to validate requests without database lookups. Scalability motivated our choice of a fully stateless backend (SessionCreationPolicy.STATELESS). This design requires every request to include authentication data, adding slight bandwidth overhead, but it enables horizontal scaling without sticky sessions or shared session storage. Maintainability was achieved through a strict layered structure (Controller → Service → Repository → Entity). While this introduces additional boilerplate and multiple classes for simple features, it ensures business logic stays isolated from HTTP and persistence concerns, making unit testing and future refactoring significantly easier. Interoperability drove the decision to expose a REST-based JSON API consumed by a standalone React SPA. Although this approach requires DTOs and careful separation between internal entities and API contracts, it allows the frontend to evolve independently and supports future integration with mobile or external clients. Finally, we prioritized data integrity by choosing PostgreSQL with JPA/Hibernate. The relational model's rigid schema was preferred over flexible NoSQL stores because our domain relies heavily on enforced relationships across Studies, Participants, Tasks, and Artifacts, preventing orphaned data and ensuring consistency through foreign key constraints.

The system relies on two primary connectors: REST (HTTP/1.1) and JDBC/Hibernate. REST communication is implemented with Spring Web MVC on the backend and Axios on the frontend. We selected REST instead of alternatives such as SOAP—too verbose and heavyweight for our CRUD-dominated workflow—or GraphQL, which introduces schema management complexity without providing substantial benefit for our relatively straightforward data access patterns. REST is stateless, cacheable where appropriate, and aligns naturally with resource-based modeling such as POST /api/tasks/{id}/complete. The second connector is the database access layer, where we use the PostgreSQL JDBC driver wrapped by Spring Data JPA/Hibernate. Although ORM tools introduce some performance overhead and require understanding lazy vs. eager loading behaviors, they eliminate significant boilerplate SQL, enforce object–relational mappings, reduce the risk of SQL injection, and manage connections efficiently. Compared to raw JDBC, this approach allows faster development and more consistent data handling while still providing transactional reliability.

The system employs a Client–Server architectural style combined with a strict layered design within the backend. In the Client–Server model, the React frontend manages presentation logic and user interaction (e.g., dynamic sliders, responsive UI), while the Spring Boot backend performs business processing, validation, and database access. We selected this style because it supports a rich SPA experience and cleanly separates concerns between UI rendering and server-side computation. Alternative styles such as monolithic server-side rendering using JSP/Thymeleaf or microservices were evaluated but dismissed: server-side rendering would tightly couple UI and backend development, while microservices would impose operational complexity and infrastructure overhead disproportionate to the project scope. Within the backend, we enforced a strict Layered Architecture consisting of Presentation, Business, and Persistence layers. This ensures that controllers never bypass services and call repositories directly, protecting validation rules, access control enforcement, and transactional consistency. Although more flexible patterns like Hexagonal Architecture were considered, the layered approach offered the best balance between simplicity, maintainability, and enforceable separation of concerns for the system’s current size.

Subsystem Decomposition Diagram