**DESIGN DOCUMENT**

**ARCHITECTURAL CONSTRAINTS AND DESIGN DECISIONS**

1. **Why Spring Boot?**

* Ease of Development: Spring Boot simplifies the development of Java-based applications by autowiring and providing a pre-configured setup that eliminates many aspects of the configuration.
* Integration Capabilities: Spring Boot provides excellent support for integrating with other technologies and frameworks, such as JPA making it a versatile choice for building applications.
* Community and Ecosystem: It has a large community and a rich ecosystem, which means there are plenty of libraries, documentation, and support available. This reduces the risk and costs associated with development.
  1. **Why not another tool or technology?**
* Node.js/Express: While Node.js with Express could be an option for building web applications, Spring Boot offers a more structured approach for Java applications, along with support for dependency injection.
* Django/Flask: Python frameworks like Django or Flask are great for rapid development, but they may not provide the same level of support for Java-based ecosystems or microservices-oriented architectures.

1. **Why are you using React?**

* Component-Based Architecture: React's component-based approach allows for modular development, making it easier to maintain and reuse components across the application. This is ideal for creating dynamic, interactive user interfaces.
* Virtual DOM for Performance: React uses a Virtual DOM, which optimizes updates and rendering, resulting in better performance for highly interactive UIs and page switching.
* Flexibility: React's unopinionated nature allows for greater flexibility in choosing tools for routing, state management, and other functionalities (in my case my framework is Next.js), as opposed to more opinionated frameworks like Angular.
* Community and Ecosystem: With a large community, React has a wide array of third-party libraries, tools, and support, making it easier to find solutions for common problems.
  1. **Why not another tool or technology?**
* Angular: While Angular offers a more opinionated framework with built-in solutions for routing, forms, and HTTP services, it is way more complex and less flexible.
* Vue.js: Vue is a viable alternative to React but I am more familiar with React’s Next.js framework

1. **Why are you using MySQL?**

* Relational Data Requirements: The project requires structured data storage with relationships between tables, which is well-suited for a relational database like MySQL.
* Maturity and Stability: MySQL is a mature and stable database management system, widely used in the industry with robust support for transactions, indexing, and querying.
* Community and Support: Being one of the most popular databases, MySQL has a large community, plenty of documentation, and active development, which ensures reliable support.
* Open-Source and Cost-Effective: MySQL is open-source, making it a free and wide-spread technology
  1. **Why not another tool or technology?**
* PostgreSQL: While PostgreSQL offers more advanced, MySQL's ease of use, performance for read-heavy operations, and simpler configuration make it a more suitable choice for the project requirements.
* NoSQL Databases (e.g., MongoDB): NoSQL databases are better suited for unstructured or semi-structured data. Since the project involves structured data with relationships, a relational database like MySQL is more appropriate.
* SQLite: SQLite is lightweight and suitable for smaller projects or embedded applications but may not perform as well as MySQL in larger environments.

**C4 diagrams**

**C1**

Diagrama

Descripción generada automáticamente

**Level 1: System Context Diagram**

The diagram provides a basic overview of the "RaiseHub" system, a platform that allows users to create and fund projects. It shows interactions with three primary actors and systems:

1. **Admin [Person]**: An admin can perform management tasks, such as funding projects and deleting any projects in the system. Admins receive notifications via email when a project is deleted.
2. **User [Person]**: A user can create new projects, fund other projects, and view existing projects in the system. When a new project is created or a project reaches its funding goal, the user is notified via email.
3. **E-mail System [Software System]**: A third-party email service (e.g., Google email system) used for sending notifications to users and admins regarding project status updates and deletions.

The central system in this diagram is **RaiseHub [Software System]**, which facilitates these interactions by allowing users to create and fund projects

**C2**

Diagrama

Descripción generada automáticamente

The diagram provides a breakdown of the main containers that form the "RaiseHub" system and how they interact with each other. It includes user roles, containers for different system functionalities, and external dependencies. Here’s an analysis of each component in the diagram:

**Actors:**

* Admin [Person]:
  + Can view, fund projects and delete any projects in the system.
  + Interacts with the system through the user interface and receives email notifications about certain events, such as project deletions.
* User [Person]:
  + Can create new projects, fund projects, and view existing projects.
  + Receives email notifications for events like project creation or reaching funding goals.

**Containers:**

1. Single-Page Application (SPA) [Container: React]:

* A front-end application built with React, providing all the functionalities for users to interact with the platform. This includes creating projects, funding projects, and viewing existing projects.
* Communicates with the API Application through HTTP API calls to fetch or update information.
* Provides a dynamic and responsive user experience.

1. API Application [Container: Spring Boot]:

* The backend service responsible for managing the business logic of the application.
* Handles requests from the Single-Page Application and processes information related to users, projects, and payments.
* Interfaces with the Database to store and retrieve data using JPA (Java Persistence API).
* Sends notifications via the E-mail system to inform users about various events.

1. Database [Container: MySQL Relational Database]:

* Stores all persistent data, such as user details, project information, and payment transactions.
* Interacts with the API Application, which retrieves and updates information through JPA.

1. E-mail System [Software System]:

* A third-party service (e.g., Google email system) used to send email notifications to admins and users.
* The API Application triggers email sending based on certain events, like project creation, deletion, or when a funding goal is reached.

**Software Principles Applied:**

1. Single Responsibility Principle (SRP):

* Each container has a distinct and well-defined responsibility. For example, the API Application handles business logic, while the Single-Page Application is responsible for the user interface.

1. KISS (Keep It Simple, Stupid):

* The architecture is kept simple by dividing the system into clear and manageable components. There’s a straightforward flow of data from the user interface to the backend and the database, with notifications managed separately.

**C3**

**Diagrama

Descripción generada automáticamente**

The diagram provides a detailed breakdown of the internal structure of a specific container, focusing on the components within the API Application. It highlights the key components responsible for handling various functionalities (User, Project, and Payment management) and their interactions with the database.

**Main Components:**

1. Single-Page Application (SPA):

* The front-end SPA interacts with the backend by making API calls to various controllers. These controllers expose different functionalities of the system, such as managing users, projects, and payments.

1. Controllers (UserController, ProjectController, PaymentController):

* The Controllers serve as entry points for API requests. Each controller manages a specific domain:
  + UserController: Manages user-related functionalities such as user creation, deletion, and information retrieval.
  + ProjectController: Manages project-related functionalities, including creation, deletion, and information retrieval.
  + PaymentController: Handles payment-related operations, such as processing payments and retrieving payment information.

1. Service Layer (UserService, ProjectService, PaymentService):

* The Service Layer encapsulates the business logic for each domain. It acts as an intermediary between the controllers and the repository layer:
  + UserServiceImpl: Implements business logic related to user management.
  + ProjectServiceImpl: Manages the business logic for project-related operations.
  + PaymentServiceImpl: Handles the business logic for payment processing.
* By using service interfaces (e.g., UserService, ProjectService, PaymentService), the system adheres to the Dependency Inversion Principle, where higher-level modules depend on abstractions rather than concrete implementations.

1. Repository Layer (UserRepository, ProjectRepository, PaymentRepository):

* The Repository Layer handles direct interactions with the database. It is responsible for reading from and writing to the database:
  + UserRepositoryImpl: Manages data access for user entities.
  + ProjectRepositoryImpl: Handles data operations for project entities.
  + PaymentRepositoryImpl: Manages data access for payment entities.
* This layer abstracts the data access details from the service layer, following the Single Responsibility Principle (SRP), where the repository’s sole responsibility is data management.

1. Database [Container: MySQL]:

* The Database is the persistent storage for the system, storing all user, project, and payment data.
* Each repository interacts with the database to perform CRUD operations (Create, Read, Update, Delete).

**Architectural Flow:**

* The Single-Page Application interacts with the Controllers, which handle incoming requests and delegate processing to the appropriate Service Layer component.
* The Service Layer performs business logic and coordinates with the Repository Layer to access the necessary data.
* The Repository Layer executes data operations on the Database.
* This structure follows a layered architecture, where each layer has a distinct role, enhancing maintainability and scalability.

**Application of Software Principles:**

1. SOLID Principles:

* Single Responsibility Principle (SRP): Each component has a single responsibility. For example, the UserController only handles API requests related to users, while UserServiceImpl contains the business logic for user operations.
* Open/Closed Principle: The system is designed to be open for extension but closed for modification. New functionalities can be added without changing the existing code, following the interfaces and service structure.
* Liskov Substitution Principle: The use of interfaces ensures that any implementation of the interface can be used without altering the client code, maintaining consistency.
* Dependency Inversion Principle (DIP): Higher-level modules (controllers) depend on abstractions (services) rather than concrete implementations, promoting loose coupling.

1. DRY (Don't Repeat Yourself):

* Common functionalities, such as data access, are centralized in the Repository Layer, preventing code duplication across services.

1. KISS (Keep It Simple, Stupid):

* The architecture remains simple by organizing components logically and maintaining clear responsibilities. Each layer performs specific tasks, making the overall system easy to understand.