[Año]

Architecture/design exercise - King

CRISTIAN VEGA SÁNCHEZ

Contenido

I Scope of the document

The purpose of this document is to provide a detailed analysis of software requirements, in order to provide an accurate overview of the architecture design.

1. Requirements and first approaches

1.1. Requirements

The back end application for a specific game has the following requirements:

1. Save and retrieve user progress,

2. Retrieve 10 top scores,

3. Add and remove friends in game with the option of seeing the highest score,

*** Important:** It should manage a lot of concurrent connections and spikes during game events.

1.2. Bounded context or micro services

Applying the divide and conquer strategy, we can separate these requirements into 3 contexts or topics, every user should have this components:

- Progress
- Scores
- Friends

*** Important:** Inside friend we have to be able to get the scores.

1.3. Required Endpoints – API Rest:

According to the requirements we can quickly see that we require the following **five end points**:

Progress:

1. **SaveProgress – PUT:** Save all the progress information related to the user.
2. **RetrieveProgress – GET:** Return all the progress information related to the user.

Scores:

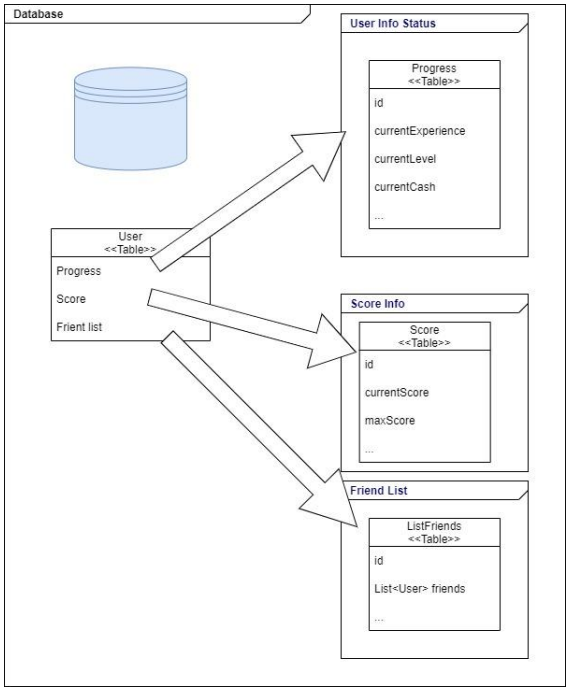
3. **RetrieveTop10Scores – GET:** Return the top ten highest/top scores.

Friends:

4. **AddFriends - POST**: Add friend to current friend list.
5. **RemoveFriend – DELETE:** Remove friend from current friend list.

1.4. Data to be stored:

A first approximation to what our database will have to store in the future, with example of maybe needed fields:



First generic DB content

A Users has:

- Progress information about current user status, like current level in game, current cash/money …etc. Information associated with the status of the user with the game.
- Score information about the points obtained on the game.
- List of friend (Users) associated with the user, list because can add or deleted them.

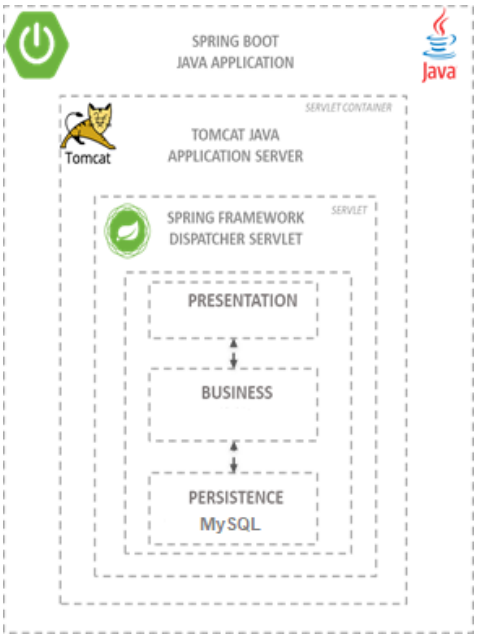
***Note:** It's a very basic initial representation of the data to be stored, basically to land the idea little by little. To different from final approach.

2. Arquitecture proposal

2.1. Backend

The logic tier, will be implemented with an API REST using Spring Boot Application totally decoupled from the frontend.

Spring Boot simplifies the deployment of Java Web Applications (Traditionally as servlets) by integrating the Java application in a single Spring based Dispatcher Servlet, and deploying it in a lightweight Java Application Server embedded in the application itself. In this case, default Tomcat Java Application Server is used.



Spring Boot application package

A good practice is use OpenAPI/Swagger documentation generation in order to synchronize the API definition with the implementation in both backend and frontend.

Top Down API Design

By doing this in this way, frontend developers can start to work with a mock of the API just by starting a mock server generated with the API specification (e.g., Postman) and can speed up the development of the communication layer by generating automatically the communications services.

2.2. Database

The database level will be developed with MySQL. Since we can see a quick view of the relationship between the different users as well as the list of friends, I think a relational database is the best idea. Another advantage of using MySQL is that each service can easily create here own Schema of the database by using a file .sql file creator.

Database as code

Spring Data JPA repositories can be used to implement database access. Spring Data JPA simplifies database access by defining repository interfaces, which are automatically implemented at runtime by Spring Framework providing standard CRUD operations, and simplifying basic query operations.

Shared database versus Database per service

If all the micro services/database attack to the same central repository (shared database):

We have to take care of multiple ready peaks on same DB (unique), one solution for this maybe to install and intermediate **cache** DB like Redis.

If the problem is related to multiple writing simultaneous on same DB we have to use a **Sharding** strategy to split storing dataset in multiple databases.

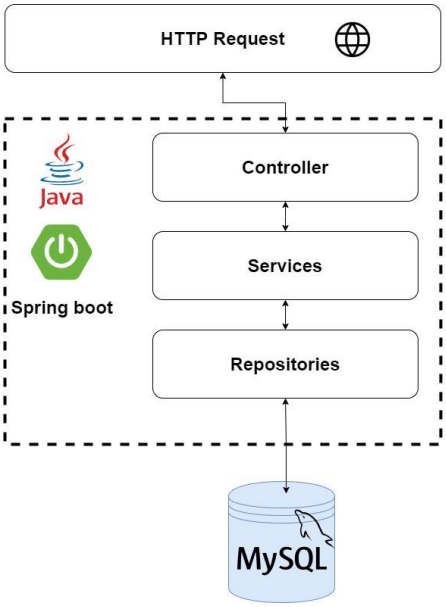
In this particularly case I think is better to apply **database per service** because that pattern allows us to scale the application in an independent way.

3. Design proposal

3.1. Folder Structure

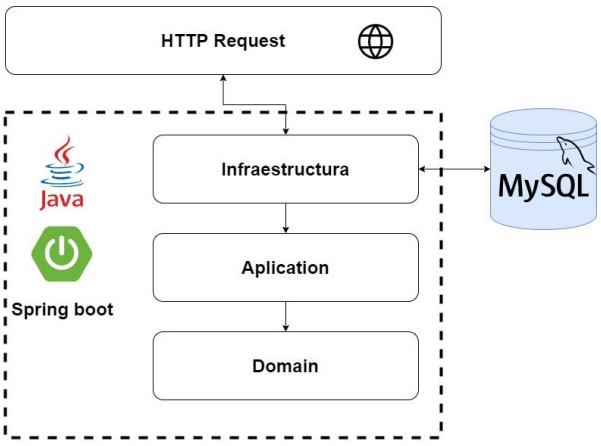
Proposal 1 – Spring boot 3 layers API Rest:

- **Controller**: Entry point of the application, receives the http requests and executes the services for each case.
- **Services**: Part where all the calculation and logarithmic functions are performed as well as calling the repository.
- **Repositories**: In charge of generating an interface between the database and our application. The queries and entities are also defined here.



Proposal 2 – Domain Driven Design

- **Infrastructure**: 3th party frameworks and API Controller (Input layer).
- **Application**: Place where all use cases are defined
- **Domain**: Definition of all aggregates and domains.



Domain-driven design proposal

3.2. Software Improvements

In order to perform the source code analysis following techniques can be apply:

- Static source automatic analysis to obtain standardized metrics.
- Use best practices of the different frameworks to be used.
- Try to use design patterns as much as possible to avoid reinventing the wheel
- Check clean code principles.
- Introduce Solid principles.