FLARETASK

SHORT DESCRIPTION

1. Table of contents

1. Table of contents 1
2. User Story 1
   2.1. Task Description 1
   2.2. Requirements: 2
   2.3. Hints: 2
3. Development: 3
   3.1. Technologies and libraries: 3
   3.2. Design: 3
      Database and model structure: 3
      Project structure: 4
      Endpoints: 4
4. API: 5
5. Running the application: 6

2. User Story

2.1. Task Description

A Java REST API web application.

During application start it should load 20 newest (by date) featured questions from StackOverflow.com API:

https://api.stackexchange.com/docs/featured-questions

The result of the call should be stored into the database, results from previous runs should be removed. Make sure your database schema is optimized for the data access scenarios described below.

The application should provide REST API to query the stored data. Feel free to design the URL schema as you see fit. Provide input parameter validation and error reporting as you see fit. JSON should be used as data format. The following endpoints should be present:

- HTTP GET endpoint to get all the questions in the database
- HTTP GET endpoint to retrieve a single question by id
- HTTP DELETE endpoint to remove a single question by id from the database
- HTTP GET endpoint to retrieve all questions that have specified tags

- The following fields should be returned for each question:

- `id` (`Number`)
- `tags` (`Array` of `String`s)
- `is_answered` (`Boolean`)
- `view_count` (`Number`)
- `answer_count` (`Number`)
- `creation_date` (datetime in ISO8601 format as `String`)
- `user_id` (`Number`) of the user who had asked the question

In addition, there should be a HTTP GET endpoint that will return details about a user by id, calling (proxying) the following endpoint:

- https://api.stackexchange.com/docs/users-by-ids

The result of the call should be cached in-memory in a way, that multiple calls to the endpoint with the same id will not lead to multiple calls to the StackOverflow API.

The following fields should be returned for the user:

- `user_id` (`Number`)
- `creation_date` (datetime in ISO8601 format as `String`)
- `display_name` (`String`)

2.2. Requirements:

- Code and docs should be committed in a public Git repository (GitHub, GitLab, Bitbucket)
- Java 8+
- Use any REST API framework (Dropwizard, Spring Boot etc)
- Sufficient unit test coverage is a must (having integration tests is a plus!)
- Use a database of your choice. It can be traditional RDMS or NoSQL. In-process (embedded) database is ok.
- Use a build system of your choice: Gradle, Maven etc
- Feel free to choose the final artifact format: it can be anything from single jar file to a Docker container
- Code should be accompanied with a short description of:
    - Architecture decisions and your reasoning behind them
    - If you decide to use libraries, please outline how and why you picked them
    - Instructions on how to build and run the project

2.3. Hints:

Architectural and code quality (general software development best practices like clean code, separation of concerns, ...) are key to success.

3. Development:

3.1. Technologies and libraries:

At choosing the frameworks and core technologies, my main goals were:

- portability: because at the end of the day this is a test
- simplicity: the solution should be adjusted to the scale of the problem
- scalability and maintainability: the app itself and the used technologies should be easy to change

Based on the reasons above I chose the Spring Boot framework as the core technology of the application. It provides out-of-the-box solutions for the given task with the option of customisation and the possibility of scaling.

As for the DBMS, I preferred in-memory solutions because of the portability and the scale. I decided to use H2 based on that it is lightweight and has good compatibility with Spring.

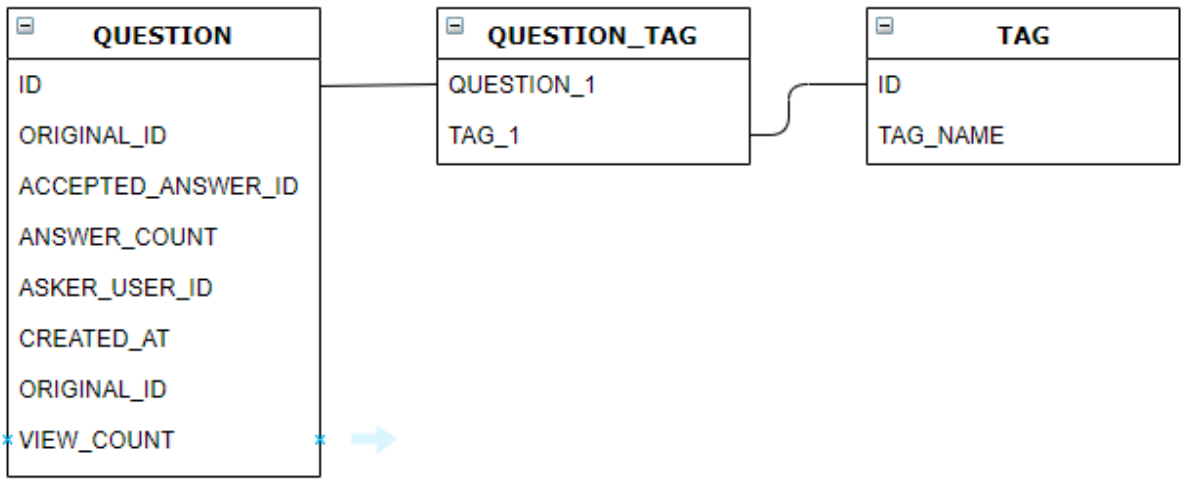
Other technologies were picked by the good integration with the framework and the following reasons:

- MAVEN: I have experience in it
- SpringData with JPA: easy-to-handle, provides fast development
- Lombok: spares writing boilerplate code
- Junit and Mockito: provide a complete solution for unit tests and I have experience with them

3.2. Design:

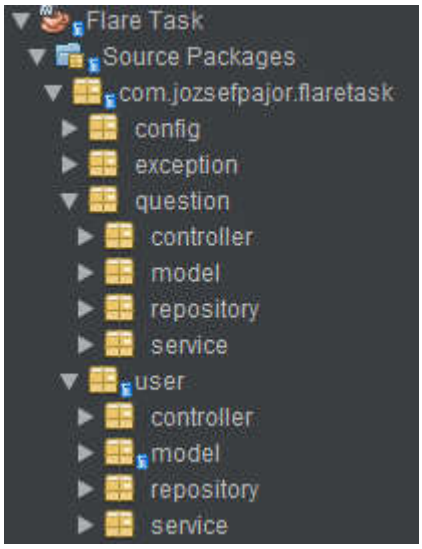
Database and model structure:

For only the Questions should be stored, the model structure is quite straightforward. Tags are separate business objects in Stackexchange API, therefore they have their own model class, which indicates the need of the many-to-many connection as entities and a join table in the DB.



Project structure:

Following the recommended vertical slicing of microservices, I decided to split the structure by business units (questions and users). Inside the units, the project follows the “standard” Spring Boot REST API packaging.



During the development, I have worked around interfaces, which (together with the project structure) ensures easy modularization and change of technologies.

Endpoints:

I created the service endpoints according to the common best practices of REST APIs. These suggested implementing the endpoint, which retrieves all questions that have specified tag, as a filter of the endpoint of all questions.

4. API:

The following endpoints are present (localhost:8090):

Method	URL	Content	Function	Parameters	
GET	/api/v1/questions?tag={tagname}	application/json	Returns all stored questions (with filter)	tag: String	filter for tagname
GET	/api/v1/questions/?tag={tagname}	application/json			
GET	/api/v1/questions/{id}	application/json	Returs the data of the specified one question	id: Long	Business ID of the question (question_id in Stackexchange API)
DELETE	/api/v1/questions/{id}	-	Deletes the specified one question from the database	id: Long	Business ID of the question
GET	/api/v1/users/{id}	application/json	Returs the data of the specified one user	id: Long	Business ID of the user (user_id in Stackexchange API)

5. Running the application:

After the checkout/download from the GitHub repository:

1. Execute “mvn install”: compiles the executable jar in the /target directory. Originally built with MAVEN 3.6.3
2. Run “runapp.bat” found in the root repository directory: runs the app with an open command prompt.

By default, the server listens for client calls at **http://localhost:8090**.

Hints:

- Failing to load initial data stops the server, for its main porpoise is to provide these up-to-date data.
- Successful initialization is logged with the message: “Initializing latest questions was successful”.
- To ease testing the caching of User data, the call for user data towards Stackexchange API is logged with the message: "Stackexchange api called for user data with user id: {user_id}“.
- DB files are created in /target/data directory.