GetYourGuide SE Tech Challenge Architecture Documentation

By: Niyi Omotoso.

Table of Content

Summary:

This is a simple implementation of a web application that allows users to search for activities. The user is able to see details of the activities available including the event organizer/supplier.

Implementation Overview/Technology Stack

Frontend: Framework: Vue.js, Package manager: npm.
API: Framework: SpringBoot, Package manager: gradle.
Database: MySQL.
Containerization Engine: Docker.

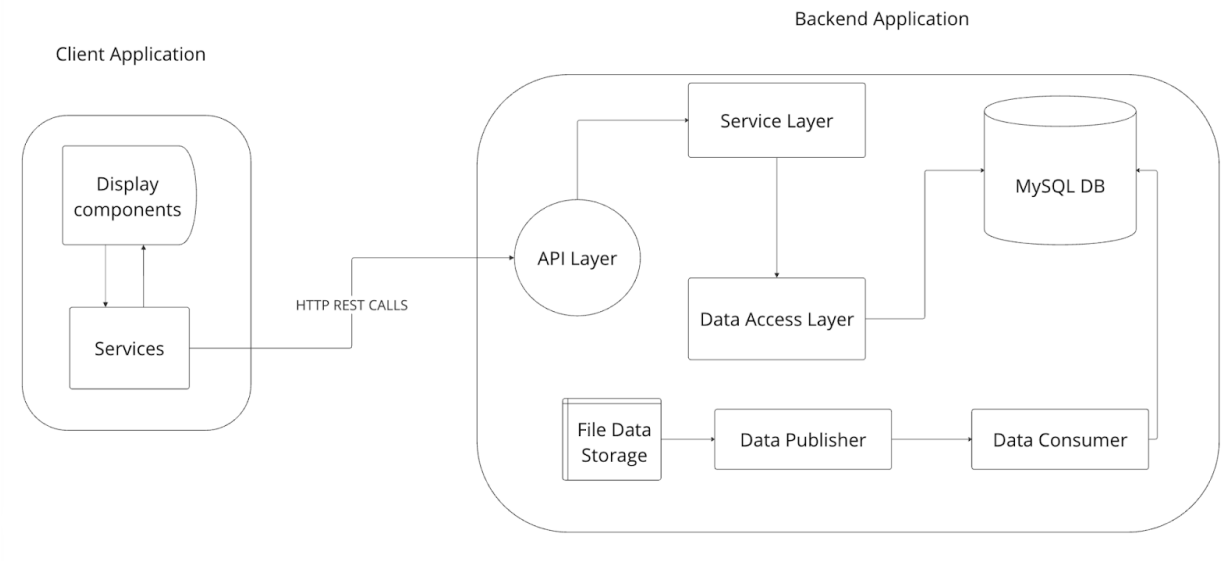

Fig .1 high level system architecture of the activity search web application

Design Approach

Backend Architecture:

I went with a layered architecture to implement this backend system. I added these layers to organize the code, separate concerns, and ensure the application is maintainable and scalable. Here's an explanation of the different layers, from the API layer to the database layer:

1. **API Layer** (Presentation Layer): This is the entry point of the backend and is responsible for handling incoming HTTP requests from clients. Its main tasks include:
    - Parsing incoming requests.
    - Validating input data.
    - Orchestrating business logic by calling appropriate services for activities.
    - Formatting and sending responses back to the client.

2. **Service Layer** (Business Logic Layer): The service layer contains the application's core business logic, even though it is pretty simple and lean for this project. It handles the processing of data and implements the application's rules and workflows. This layer is responsible for:
    - Coordinating multiple data sources by fetching from the appropriate data repo.
    - Applying business rules and mapping the data to the DTO.

3. **Data Access Layer**: The data access layer is responsible for interacting with the database. It abstracts away the complexities of database interactions and provides a clean interface for the rest of the application to work with data. Here I used Hibernate, the Java JPA implementation for persistence. Responsibilities include:
    - Retrieving and updating data from/to the database.
    - Executing database queries.
    - Managing database connections and transactions.

4. **Database Layer:** This is where the actual data is stored and managed. My choice is MySQL. The reason for choosing MySQL for this project is the ease of use and flexible administration. This includes the ability to run queries, joins, case-insensitive search, all still with an incredible performance and room for scalability in the future.

Notable Addition: There is an interesting sub-layer I built into the backend application to parse the data in the file into something useful in the mySQL database. This is happening over a layer of abstraction that can allow me to switch this data source for something else (e.g a messaging system) in the future.

These layers work together in a structured manner to create a separation of concerns and facilitate the maintenance, scalability, and modularity of your application. I tried to design the interactions between these layers carefully, ensuring a clear flow of data and responsibilities while adhering to best practices for software architecture.

Frontend Architecture:

Similar to the backend, the frontend is layered as well,

Component Layer: I organized the frontend logic into components, self-contained units of UI that can be reused throughout the application. Each component encapsulates its template, JavaScript logic, styles and life cycle hooks. The component is organized into a tree, where parent components contain some other child components. This hierarchy enables a modular and maintainable approach to building the UI.

API Calls/Service Layer: Since the app needs to communicate with backend APIs to fetch and update data, I used the built-in `fetch` API to make HTTP requests. Data retrieved from APIs and stored in the local component state.

Style Layer: Since Vue.js supports various approaches to styling, including traditional CSS, I went with the traditional styled option for convenience and to implement the basic layout that presents the data as seen.

Implemented Features

Frontend

1. By default, users see a list of activities.
2. User is able to search for activities by keyword that match the activity title (case-insensitive).
3. User is able to search for activities by keyword that match the supplier name (case-insensitive).

Backend (Functional and Non-functional)

1. By default, API endpoint returns a list of activities based on a default limit.
2. Clients can query activity by keyword that matches either the activity title or the supplier name (case-insensitive).
3. Admin can trigger data population for newly added activities and suppliers in the json file.
4. Unit test coverage for business logic.

5. Integration test coverage
6. Containerization of the modules for easy run on the local environment.

Drawbacks and Future Improvements

Frontend

- There is no test coverage for the FE logic
- No pagination logic yet to better segment the results from the backend.
- No end to end test covering both FE and BE to ensure usability acceptance.

Backend

- No full text search support.
- No fuzzy search support.
- No indexing for faster search on activity title or supplier name when the data size grows.

There are many things that can be improved on both FE and BE. For example, moving a Data layer to a data store such as **ElasticSearch** can simply take care of all the search drawbacks listed above and even more including big data.

Also having a dynamic algorithm for selecting the default results shown to the user based on some parameters determined by the business will be a good improvement.

Ultimately, by using a layered architecture, these future improvements should be easy to plug in since the system is layered and the concerns adequately separated.