Eureka

Synopsis

We intended to create a distributed event browsing and management application using open source APIs from TicketMaster. The user can browse, search, and save events as interested and will also be able to discover events based on our proprietary algorithm that suggests popular events from users interested data.

The application is for users who want an easy glance of all the events that is currently being held near them and save them for booking it later. The events could be concerts, stand ups, college events, etc.

The application will have a sign-up section and a username and password based authentication mechanism. Once logged in the application should display events nearby to the location where the user resides from their IP Address. The user will also have the capability to search events based on keywords e.g. to show all events by *Coldplay*, and can save events they like/want to go as interested. The application will also suggest popular events based on the users location and other users interested events within the location radius.

Technology Stack

- **Spring Boot:** Spring Boot for standalone applications that can run using embedded tomcat servers, without the need to create external webservers. Easy to build micro-services with auto-configuration and inbuilt frameworks for developing REST APIs quickly with lots of out of the box solutions by Spring Cloud.

- **Hibernate:** Used for object relational mapping with the database. Makes retrieving/persisting data easier using objects and comes out of the box with Spring Boot.

- **PostgreSQL:** It is used to store the application data like users and events data in a structured way. It is a tried and tested relational database that has a large open source community, and powerful index and partitioning mechanism for storing relational data and can be scaled.

- **Netflix Zuul:** Used for dynamic routing, monitoring and acts as the application gateway to forward requests to micro-services. The client only needs to interact with the proxy server. Spring Boot has a nice integration with Zuul Proxy which is why we used it in the application.

- **REST API:** The microservices uses a REST architecture to interact with web services with JSON data. REST architecture is widely used and it is an easy way to create fault tolerant distributed systems.

- **Maven:** Used for source control and handling dependencies of the project.

- **Docker:** Used for containerisation of the whole application and can be built and deployed anywhere with just a single command. Takes care of the entire application setup and running it.

- **Swagger:** Used for documentation of the APIs. The users can use the swagger docs to see the request and response and try out the APIs. (Only implemented for Authentication Service).

System Overview

The main components of the application are

- **Authentication Service:** The authentication service is the brain of the application. It has two main functionalities.
    - **Authenticate:**
        - A **Sign Up** endpoint for user sign up.
        - A **Login** endpoint for the user to sign in to the application. Once authentication is successful it sets the users context into spring authentication and then generates a JWT Token that is valid for 5min. This JWT Token acts as a bearer token that needs to be passed in the ***Authorization*** header for requests that are secured.
            - ***Key***: *Authorization*, ***Value***: *Bearer <JWT Token>*
        - All the requests that need to be authorized passes through the applications custom authentication filter which checks the JWT token validity and user details.
    - **Reverse Proxy:**
        - It also acts an application gateway to the application. All the incoming http requests passes through this service.
        - It uses Netflix's Zuul proxy server to forward and delegate requests based on the defined routes. Any incoming requests matching the route will be forwarded to the respective services. Hides the other servers and prevents from DDoS attacks. Can act as a load balancer for the application.
        - It also implements a pre filter to forward the users ID to the other services so that the services also has the users session information.
        - All the services responses are then returned to the client.

- **Geo-Location Service**: This service fetches the geo data from an Open Source API that gives the geo data based on the IP Address. It exposes an endpoint where the client can send their IP and returns the geo data like country, city, latitude and longitude. This endpoint sits under authentication. The Bearer token needs to be passed in the Authorization header.

- **Events Service:** This service is responsible for fetching and processing event data from TicketMaster API. It exposes multiple endpoints for event discovery, search and saving interested events.
    - **Event Discovery:** This POST REST endpoint is responsible for displaying events based on the clients IP Address. It communicates with the *Geo-Location* service to get the latitude and longitude and then use these attributes to fetch the events from the TicketMaster API. The communication between Geo-Location service and TicketMaster API is done using *Apache HTTP Client.* It parses the event data and picks only the necessary attributes and returns a response with list of events and also the pagination information. The API allows for pagination. Limit and Offset can be sent in the request for pagination.
    - **Event Search:** This GET endpoint allows users to search on a particular keyword and displays events based on the search key. This again makes an HTTP request to TicketMaster API and passes the search key, gets the data and parses it and returns the output.
    - **Events Interested:** This allows user to save the events that he wants to go or likes as interested. It's a wish list of the events they want to attend. It's a POST endpoint and takes a list of event IDs and saves the event, venue and the interested event and user mapping details in the database. It saves these following entities in the Postgres DB
        - **Events:** Event ID, Event Name, Event URL
        - **Venues:** Venue ID, Venue Name, Event ID, Latitude, Longitude
        - **Interested Events** (Mapping Table) **:** Event ID, User ID

- **Event Suggestion Service:** This service uses a custom algorithm to fetch events based on the current users IP and other users event preferences and location details of the venues to show popular events which saved in the database. It uses Postgres earthdistance and cube extensions to suggest popular events. It also takes into factor the number of times an event was marked as interested and ranks the events. It is a GET endpoint that takes an IP address returns the popular events details.

- **Common:** This is a common jar that has all the utility methods and data transfer objects that is shared by all the services.

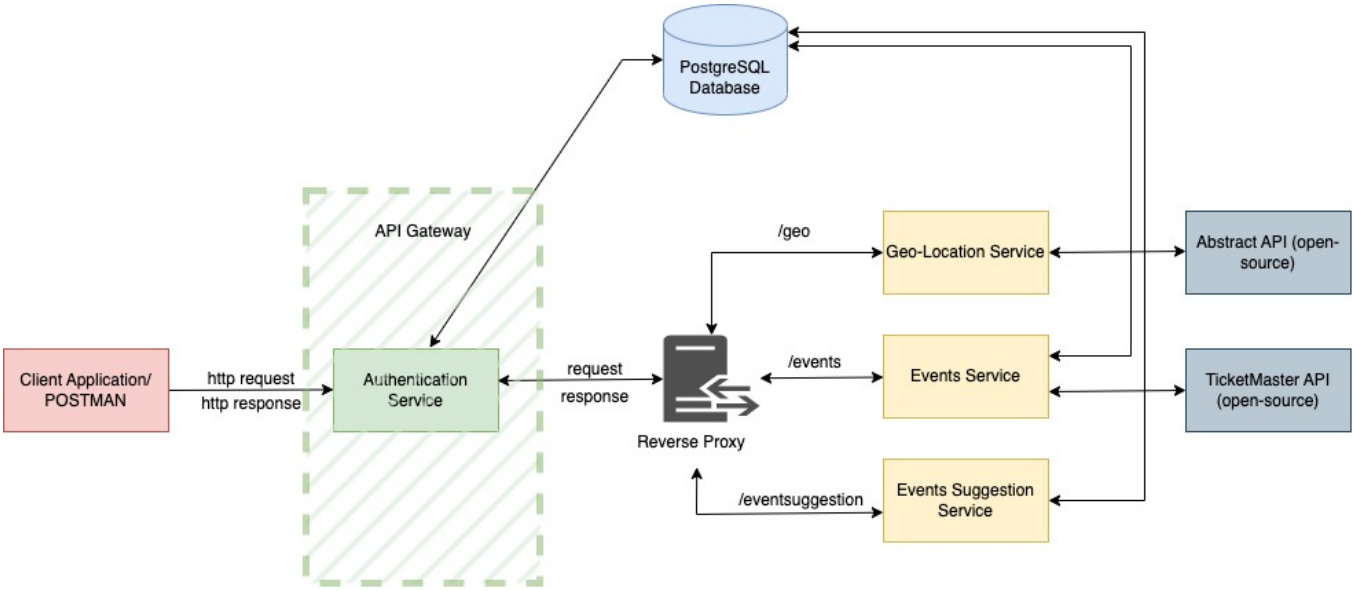

Fig 1. System architecture

Application flow

1. The client/user to use the application first needs to ***Sign Up***. The client sends a sign up request that goes to the authentication service. It takes username and password and saves the users details in the database.

2. Once the sign up is done, the client needs to authenticate into the application using username and password. The request goes to the Authentication Service where it authenticates the user and checks for the username and password and generates a ***JWT Token*** which is valid for ***5mins*** and saves the User details in Springs Security Context. The JWT Token contains the user ID, issue date and the expiry date.
    a. Both the Sign Up and Authenticate API is ignored from the authentication filter.

3. Once authenticated the client can then interact with the Geolocation Service, the Events Service and the Event Suggestion service.

4. The client directly interacts with the authentication service. All the incoming http request from the client passes through this service. The authentication service first checks if the incoming requests should be filtered for authentication, if yes then it checks the Authorization header and extracts the token and checks the validity. If the token is expired it throws an ***UNATHORIZED 401*** error.

5. The Zuul Filter checks the incoming http request sever path to see if it contains routes that are defined. If there is a match it then forwards it to defined service.
    a. Requests with ***/geo*** gets routed to Geo-Location Service running on its defined port.
    b. Requests with ***/events*** gets routed to Events Service running on its defined port.
    c. Requests with ***/event-suggestions*** gets routed to Event Suggestion Service running on its defined port.

6. All the responses sent from each services gets routed back to the Authentication service and it returns the response to the client.

7. The Event and the Event Suggestion service
    a. Makes an external http call to open source APIs to get the desired data.
    b. They also connect to the Postgres DB to store and retrieve data.
    c. These services directly interact with the Geo location service without the reverse proxy to get the latitude and longitude from it.

The system is designed in such a way that it can easily be scaled to meet large amount of traffic loads. This is achieved with the help of Netflix Zuul that can act as a load balancer. The services can be replicated to meet the traffic requirement, and based on the load balancer rules the traffic can be split between these services.

This gateway also provides additional security and hides the internal services to the external users thus preventing from DDoS attacks.

The entire application is containerised and can deployed easily with just a single command. Services can be replicated by adding a new container and changing the ports. If any of the services goes down it automatically gets restarted.

Logs are used to track errors and for debugging the application. A Global exception handler is used in all the services to delegate the errors and handle all at one and can be updated easily if required.

The database can be indexed for faster retrievals and can be partitioned based on certain conditions to make the data storing and retrieving faster. Postgres DB can also be replicated to meet high loads if needed.

Reflections

Key Challenges faced

- Forwarding the users session state to the other services.
    - This was achieved by creating a custom Zuul Pre filter to inject the users ID into the request header and modify the request and then send it to the other services.
- Setting up the database during docker compose.
    - The official Postgres image allows us to place SQL files in the */docker-entrypoint-initb.d* folder, and the first time the service starts, it will import and execute those SQL files. An init.sql was added to the root folder and under volume tag the command was executed.
- Implementing Reverse Proxy
    - This was achieved by using Netflix Zuul which has an integration with Spring Cloud. It allows any service to act as a proxy server and forward request based on the routes.
- Implementing the suggestion algorithm using location and distance
    - This was achieved by using Postgres earthdistance and cube extensions that allows to treat latitude/longitude as points and then compute distance between two points.

If we could start the project again we would first finalise the tech stack and the database schema and the application design structure before jumping to the implementation. We saw that mid-way during the implementation the schema structure had to be changed to get the desired data and thus some of the services had to be reworked again to fix it. We could have also gone with Redis to store the users authentication session as a cache which would be more robust and fault tolerant to errors and the sessions can be managed at a central place.

Building a micro-service architecture application using Spring Boot and Maven was fast and easy to setup. Spring Boot comes with its own embedded server making it easy to just run the applications anywhere. Spring Boot also has a lot of out of the box autoconfigurations for security, database connections, and creating REST APIs. It comes with starter dependencies which provides everything to build a web application. Maven makes it very easy to add and remove dependencies and manage the project. Maven downloads the artifacts hosted on Maven Central. Some of

the limitations of Spring Boot is creates a lot of unused dependencies which are not used. The starters comes with a lot of inbuilt dependencies that are never used thus not providing a fine grain control.

Hibernate was used as a layer between the Web Application and the database. Hibernate makes it easy to retrieve and persist data using java objects. It however degrades the applications performance compared to native JDBC queries as it needs to map it to an object and it does not allow for multiple inserts at the same time.

Dockerising the application allows for faster testing and deployment of the application. We also found out that we could setup the database during docker-compose and create all the required tables and initial scripts. Dealing with volumes and networks is bit tricky with dockers and data persistence is complicated.

We went with Postgres database as we wanted to store structed relational data. PostgreSQL also has a very large open source community and is a very powerful database. It has lot of inbuilt extensions and techniques for partitioning and indexing the data for faster inserts and retrievals. The downside being if the event data structure changes it would be difficult to modify the schema structure to incorporate this. Can be scaled to certain level and after that it starts becoming to scale horizontally due to the constraints of ACID.

For Reverse Proxy we went with Spring Cloud integration of Netflix Zuul. Spring Boot + Zuul integration makes It very easy to define routes and forward the requests. It also comes with pre, post filters that can be implemented for request/response logging. A few configuration to the *application.yml* file turns the application to a proxy server.