Università di Pisa

BeetleQuest

Advanced Software Engineering - 2nd Project Delivery

2024/2025

Cosimo Giraldi
Giacomo Grassi
Michele Ivan Bruna

Index

1. Introduction

The goal of this project is to develop a web app and define its architecture for creating a web-based gacha game. So the users will be able to engage in all the standard activities found in a gacha game like: *`roll`, `buy coin`, `create auctions`, `bid`*.
All these actions will be implemented through a *microservices* architecture..

2. Microservices

The main idea was to divide a monolithic system into a series of microservices, each of which handles a specific functionality. This fragmentation allows for greater modularity and control of the system. To make the web-application more scalable, the microservices have been designed to be independent and stateless. Microservices that need to store data use their own dedicated database, which they access directly. However, if a service needs to access data managed by another service, it must use the internal API which is only accessible within the internal network.

In the following pharagraps we will examinate the implemented services, and expose their functionalities.

2.1. *Auth*

User registration, login and logout are all managed in a centralised manner by the same service: the Auth service. Which also provides helper endpoints to check the validity of access tokens, allowing authentication and authorization within the web-app.

2.2. *User*

This service is responsable for managing user's account informations. A user, once logged in, can access it's account details, modify them or delete the account itself.

2.3. *Gacha*

Gacha collections are managed by the Gacha service. It allows users to get the list of available gachas as well as information on each one of them. User can inspect the personal inventory of different players and their personal one.

2.4. *Market*

The Market service allows users to perform actions involving the acquisition of `BugsCoins` and gachas. It manages auctions lifetime and transactions in the system.

Through this service users can obtain gachas by performing two actions: buy and roll. To roll the user has to pay 1000 `BugsCoins`, he/she will obtain a random gacha from the system with a probability which depends on the rarity of the gacha.

The user has the permission to create and delete it's own auctions but can not bid to them, he/she can bid to other's auctions.

2.5. *Static*

This service is responsible for serving the static content of the web-app, like the images, the *css* and the *html* files.

2.6. *Admin*

This service provides the administrator with the necessary tools to manage the system in a controlled manner, allowing operations on users, gacha, and transactions and operation carried out in the market.
It can fetch the list of users with their associated information, performs detailed searching, modify users profile, view all the transactions carried out by a user and ispect user's auction list.
It can perform global actions on the gachas, like: add new one, modify/delete an existing one and get information on the system gachas. The service provides similar actions also on transactions and auctions.

3. Architecture with MicroFreshner

The microservices architecture defined for this project is the result of a process of analysis and detection of the smells present in the original monolithic prototype, carried out using MicroFreshner.

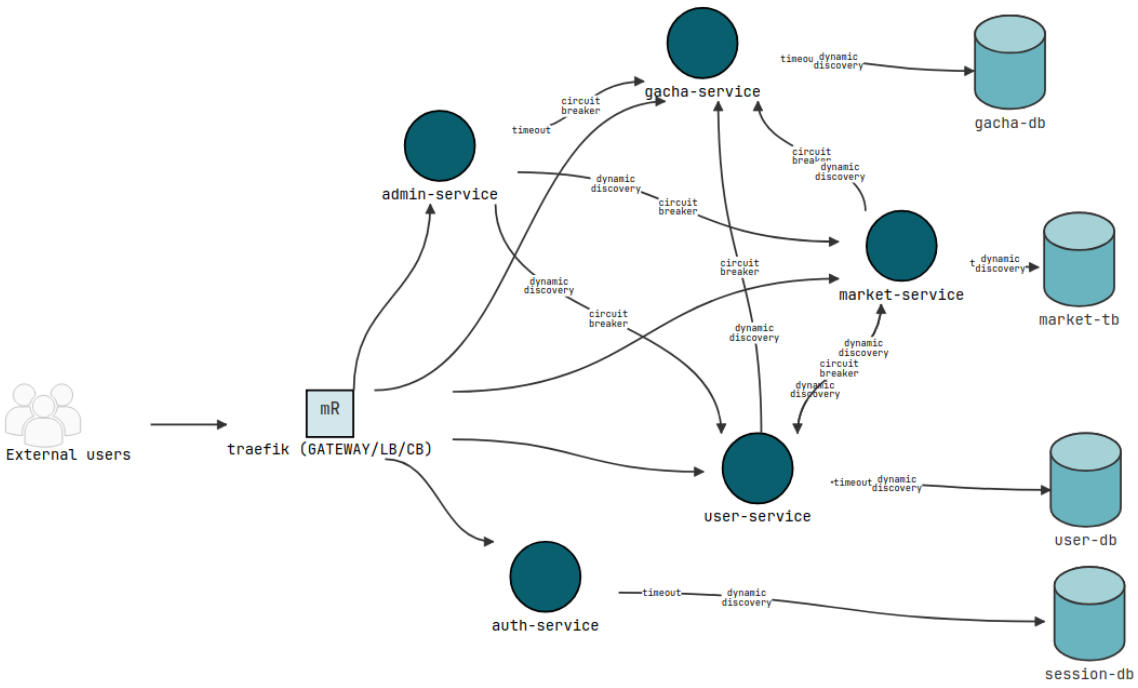

Figure 1: BeetleQuest architecture

4. Architectural Design Choices

The architectural analysis of our initial system, carried out using MicroFreshner, revealed smell between the microservices. To isolate potential failures and improve the system's resilience, we introduced Circuit Breakers (CBs).
The introduced Circuit Breakers effectively address the issues caused by continuous failures of a microservice, preventing the cascading propagation of errors that could slow down or completely halt the entire system.
Moreover, to achieve more effective control over the system, we have introduced ***Timeouts*** on database connections. This solution significantly improves resilience and reliability. If a connection or query exceeds the maximum time defined by the timeout, the system considers the operation as failed and immediately activates error-handling mechanisms, ensuring a quick response and preventing bottlenecks or slowdowns.
We have also used a reverse proxy called ***Traefik***, which acts as an intermediary between external users and the system's internal services. In this architecture, Traefik functions as an access gateway, managing and routing requests to the appropriate microservices, ensuring efficient and centralized traffic handling.

5. Interesting Flows

Now we proceed analysing a few use case scenarios, to show the flow on the backend.

5.1. Registration and login:

When a player wants to register he sends a `POST` request to the API Gateway at `/auth/register` containig the user's *username*, *email* and *password*.

- The Gateway forwards the request to the `auth` service.
- The `auth` service checks for the validity of the provided data;
- If no error occurs it creates the new user;
- It sends a request to the `user` service to create the new user data.
- The user service store the new user data in the `user DB`.
- The `user` service forward the action's status to the `auth` service.
- If `auth` service gets no error it returns to the API Gateway, a success message.

Now the user can login trough a `POST` request to the API Gateway at `/auth/login` containig the the *username* and the *password*.

- The Gateway forwards the request to the `auth` service.
- The `auth` service checks for the validity of the provided data comunicating with the `user` service.
- The `user` service checks if the user exist in the `user DB`, and return it's data to the `auth` service.
- If the user exist and the provided data is correct the `auth` service returns, to the API Gateway, a response containig a `token` that authenticates the user.

From now on we assume that all the requests contain the authentification `token` and that every microservice will obtain the `user_id` from it.

5.2. Roll gacha:

To roll for a gacha the user must send a `GET` request to the API Gateway at `/market/gacha/roll`

- The Gateway send the request to the `auth` service.
- The `auth` service checks for the validity of the `token`.
- If the `token` id is valid then the request gets fowareded to the `market` service.
- The `market` service will ask the `user` service if the user exists and it's data, then it checks if it has at least 1000 `BugsCoins`,
- If so it removes that amount of money form the user, saving the transaction in the `market DB`.
- The `market` service will request the `gacha` service to get the list of all the gachas.
- At this point the `market` service will extract randomly a gacha and, in the case that the user does not own that gacha, forward to the `gacha` service a save operation of the gacha to the user in question.
- If no error appears it returns a success message to the API Gateway.

5.3. Create auction:

A user can create an acution sending a `POST` request to the API Gateway at `/market/auction` containig the *gacha id* and the *expiration time* of the action.

- The Gateway send the request to the `auth` service.
- The `auth` service checks for the validity of the `token`.
- If the `token` id is valid then the request gets fowareded to the `market` service.
- The `market` service will check if the user has the specified gacha in his inventory, comunicating with the `gacha` service, then it will check if the *expiration time* is valid.
- Then it will save the acution in the `market DB` and set a timed event in the `timed event DB` to close the auction.
- If no error appears it returns a success message to the API Gateway.
- If no other user bid to this auction before the *expiration time*, the `market` service will automatically remove the auction from the DB.

5.4. Bid an auction:

To bid an auction a user has to send a `POST` request to the API Gateway at `/market/auction/<auctionId>/bid`, where `<auctionId>` is the id of the auction. The request has to include the amount the user wants to bid.

- The Gateway send the request to the `auth` service.
- The `auth` service checks for the validity of the `token`.
- If the `token` id is valid then the request gets fowarded to the `market` service.
- Now the `market` service will check with the `user` service if the user has the amount of `BugsCoins` he wants to bid.

- If the check passes the `market` service will comunicate the `user` service to remove the amount from the bidder, and store the bid in the `market DB`.
- If no error appears it returns a success message to the API Gateway.