UNIVERSITÀ DI PISA

A.S.E - 2024/2025
First Prototype Delivery

EzGacha Team

Gioele Dimilta
Andrea Mugnai
Jacopo Tucci

Contents

1 Introduction

This project aims to develop a microservices-based web application to collect and exchange gachas. In the second section, a brief description of the architecture and database structure is provided. Later, we present some use cases to describe the functionalities of the system. Finally, we explain how to execute the application inside a virtualized environment based on *Docker*.

2 Architecture

The architecture is based on microservices and each one implements a specific functionality. The overall structure is described in Figure 1.

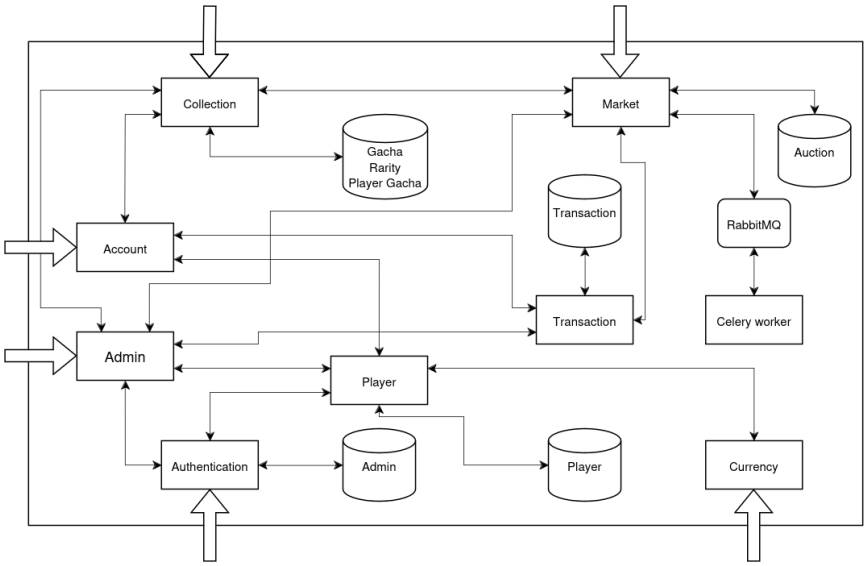

Figure 1: General architecture

A player can register himself through the *Account Service*. Then, he can login by *Authentication Service*. After that, he can perform some operations inside the website, e.g.

- Roll a gacha and check system gacha collection (*Collection Service*)
- Change password/username, retrieve his own gacha collection and check his transactions history (*Account Service*)
- Create an auction to sell a gacha and make bids on active auctions (*Market Service*)

- Purchase in-game currency (*Currency Service*)

Unlike the other services, *Player Service* and *Transaction Service* are not linked to the outside since they only manage the access to the databases (*Player* and *Transaction* tables). Players and admins can access only the API exposed by a *CaddyServer* web reverse proxy, which allows to map specific services API to the outside. The login functionality is based on jwt tokens and each service can check the token validity to accept/discard an http request.

The *RabbitMQ* message broker allows to send *tasks* to the *Celery Worker*. In order to manage the auction closing operation, *Market Service* sends a new task to the worker, which will wait until auction expires before notify the service to close the auction. Moreover, *RabbitMQ* manages tasks through a queue structure and it can re-schedule a task if the associated worker fails. Thus, using two *Celery Workers*, we can guarantee fault tolerance with respect to auction expiration.

2.1 Architectural Smells

The *MicroFreshener* architecture is described in Figure 2.

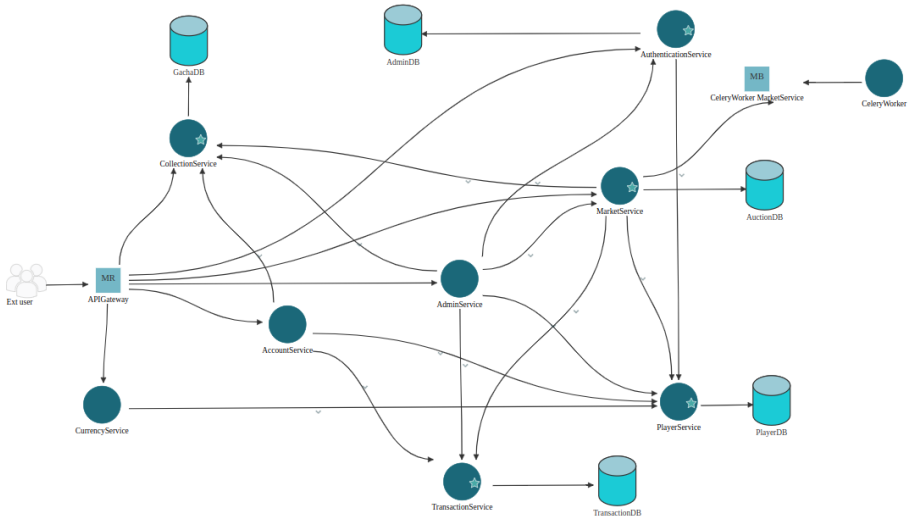

Figure 2: MicroFreshener architecture

Using the *MicroFreshener* tool, we searched for architectural smells and we solved them accordingly. We used the *circuit breaker* design pattern to handle the *wobbly service interaction smell*. No additional smells came out during analysis phase.

2.2 Database

Here is an E-R diagram of the previous monolithic database.

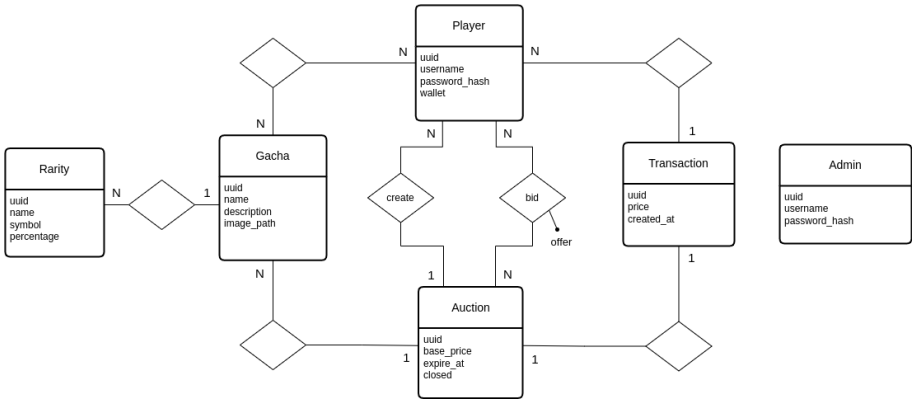

Figure 3: Database structure

The *N-to-N* relationship between *Player* and *Gacha* has been implemented with the table *Player_Gacha*. The *Bid* table expresses the *N-to-N* relationship between *Player* and *Auction*. Since the architecture is now based on microservices, the database has been splitted as follow.

Service	Database tables
Player	Player
Authentication	Admin
Collection	Gacha, Rarity, Player_Gacha
Transaction	Transaction
Market	Auction, Bid

2.3 Services

Player Service and *Transaction Service* act as database managers for *Player* and *Transaction* tables respectively. In particular, they provide basic CRUD APIs to interact with those tables, i.e. GET, POST, PUT and DELETE methods to fetch, insert, modify and delete records.

The *Authentication Service* allows players and admins to login and logout. It creates a jwt token if credentials are correct. Moreover, the admin token contains a boolean attribute *is_admin*. Admin authentication is done by *Admin Service*. In particular, when an admin wants to login, he communicates with *Admin Service*, which sends a request to the *Authentication Service*. In this way, we can keep admin functionalities separated from the player.

The *Account Service* allows players to manage their account, see their gacha collection (actually passing by *Collection Service*) and transaction history. A

player can modify and delete his account.

The *Currency Service* allows player to purchase in-game currency. A player can specify the amount of currency he wants to get and his wallet will be updated accordingly.

The *Collection Service* manages the system gacha collection and the related rarities. Also, it controls the *Player_gacha* table, which contains the infos about all the player's collections. Therefore, the *Account Service* must communicate with *Collection Service* when the user wants to retrieve his collection. The *Collection Service* provides also the *roll* functionality, which returns a gacha according to the following probabilities.

Common	**Uncommon**	**Rare**	**Epic**	**Legendary**
50%	30%	10%	8%	2%

The *Market Service* is the most complex component of the architecture. It manages auctions, bids and payments. When a player creates a new auction, the service sends a task to the *Celery Worker* in order to guarantee that the auction ends at a specific expiration time, which is 24 hours after auction creation. In particular, the worker will do a POST request to */payment* market API and the service will close the auction managing both the payment operations and gacha transfer from seller to buyer. If the payment is successfull, a new transaction record is created.

3 Use Cases

3.1 User login

The player wants to login into the application.

1. The player sends a POST request to the reverse proxy at */login* with username and password.
2. The proxy forwards the request to the *Authentication Service*.
3. The microservice checks if the user is already logged. If not, it sends a GET request to the *Player Service* at */username/*{*username*} to retrieve the player informations.
4. The *Player Service* retrieves user data from the database and sends them to the *Authentication Service*.
5. It checks if the password is not wrong and, if so, it generates a JWT session token.
6. An HTTP response containing the session token will be sent back to the proxy, which will forward it to the user.

3.2 Create auction

After login, the player creates a new auction.

1. The player sends a POST request to the reverse proxy at */market* with the required data, i.e. a gacha's UUID and the starting price.
2. The proxy forwards the request to the *Market Service*.
3. The microservice checks the presence and validity of the JWT session token from the request cookie and extracts the player's UUID from the token payload.
4. The microservice retrieves the player's gacha collection by sending a GET request to the *Collection Service* at */collection/user/*{*player_uuid*}.
5. Then *Market Service* verifies if the player owns the specified *gacha_uuid* in sufficient quantity and checks the database for active auctions of the same item.
6. If the player does not own the item or the number of active auctions matches or exceeds the available quantity, an error is returned.
7. If all checks pass, a new auction is inserted into the database with a unique UUID, the specified starting price, and the auction's expiration time.
8. After inserting the auction, the microservice retrieves the details of the created auction from the database, and a background task *invoke_payment* is scheduled to handle auction's expiration.
9. An HTTP response containing the created auction's details is sent back to the proxy, which forwards it to the user.

3.3 Gacha roll

After login, the player rolls a new gacha.

1. The player sends a GET request to the reverse proxy at */collection/roll*.
2. The proxy forwards the request to the *Collection Service*.
3. The *Collection Service* checks if JWT session token is valid.
4. The microservice randomly picks a gacha.
    - It retrieves the rarity probabilities from the database and use them to randomly choose a rarity.
    - All the gachas whose rarity matches the selected one are fetched from the database.
    - A gacha is randomly selected.

5. The *Collection Service* sends a POST request to *Player Service* at */currency/buy* in order to update the player wallet.

6. The microservice checks the database if the player already owns the gacha.

   - If the player has the gacha, the quantity is incremented.
   - Otherwise, a new record is inserted.

7. The *Collection Service* sends an HTTP response with the gacha infos to the proxy, which forwards it to the user.

3.4 Currency purchase

After login, the player purchases in-game currency.

1. The player sends a POST request to the reverse proxy at */currency/buy* with the desired purchase.

2. The proxy forwards the request to the *Currency Service*.

3. The service retrieves the JWT session token from the player's cookie and verifies its validity.

4. If the token is valid, the service sends a POST request to the *Player Service*'s endpoint */currency/buy* with the player's UUID and the purchase details.

5. The *Player Service* updates the player's wallet in the database.

6. If the operation is successful, the microservice sends an HTTP response to the reverse proxy, which forwards it to the player.

4 How to run

The application has been virtualized using *Docker*. Each component (services, databases, message brokers, etc.) is a container and we can manage them with *Docker Compose*. In order to run the application, clone the github repo and enter inside *src/* directory. Then, build the images and run the containers.

```
git clone https://github.com/pklone/ASE-project.git
cd ASE-project/src
docker compose up -d --build
```

The application is accessible from the proxy end-point at

```
https://ase.localhost
```

At the moment, no web GUI is available. However, you can use tools like *curl* to interact with the application. Check the github repo for some examples.