Advanced Software Engineering

University of Pisa, Department of Computer Science
Project Report
Academic Year 2024-2025

Marco Carollo
Francesca Funaioli
Alessia Guttadauro
Claudio Moramarco

Gachana repository

1 Introduction

The Gachana application implements a gacha game that allows players to buy gachas and sell them into auctions. The application adopts the microservice architectural pattern and its architecture is shown in Figure 1. Figure 2 shows the same architecture, but represented using MicroFreshener: when executing the analyze tool of MicroFreshener, no smells were detected. The access to each microservice is not direct, but it is mediated using an API Gateway implemented using Nginx. Each one of the microservices is associated with a database. The microservices and the databases are each deployed as docker containers. In particular, each microservice is implemented through a Python image, while the databases are implemented through MySQL docker images. The containers are then run using Docker Compose, which also defines a volume for each database in order to make data persistent when the containers are stopped. The application has been tested using a Postman collection and an automated workflow on the Github repository[1].

1.1 Microservices

The four microservices are Currency, Gacha, Market and User. The system allows for two different types of users: players and admins. The *Currency microservice* allows players to buy in-game currency using real money and to get a history of their transactions, while it allows admins to inspect all transactions and to get a specific user's transactions. A transaction here is intended as an exchange of real money for in-game currency. The *Gacha microservice* allows users to inspect the gachas they own, as well as checking which gachas they do not yet own and to roll for a gacha, while it allows admins to inspect the gacha collection, to add, remove and modify specific gachas. The *Market microservice* allows players to create an auction for a gacha they own, to see ongoing auctions, to make and accept offers, as well as seeing their own market history; it allows admins to see the details of an auction, modify it and get the whole market history. The *User microservice* allows players to create an account, login and logout, while it allows admins to get information on registered users, modify and delete their accounts.

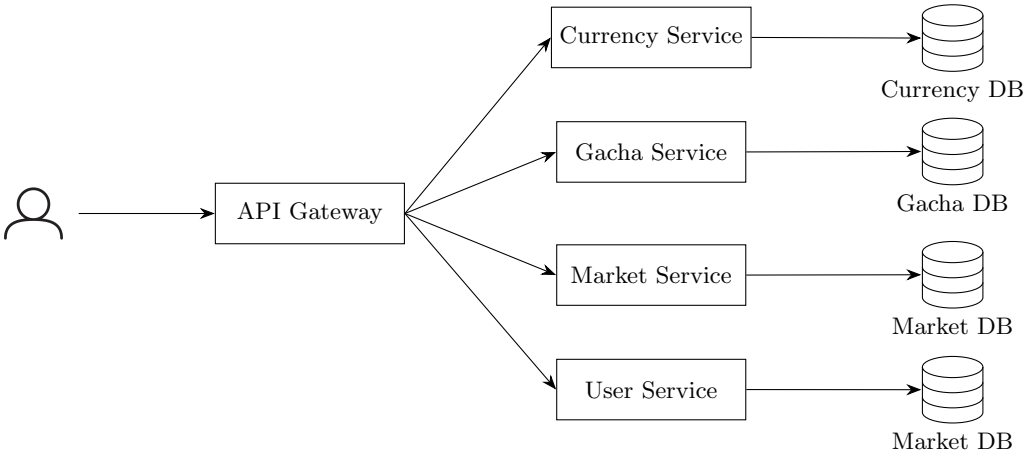

Figure 1: Gachana application architecture.

1.2 Databases

The structure of the tables contained in the databases is the one shown in Figure 3. In particular, the *Currency DB* contains the tables `Wallets` and `Transaction_History`, which respectively store information on the amount of in-game currency and on the past transactions of a player. The *Gacha DB* contains the tables `Gacha` and `Collection`: the first one stores data on all the gachas available in the application, while the second stores information about users' collections. The *Market DB* contains the tables `Market`, `Market_History` and `Offers`, which respectively store data on ongoing auctions, the offers related to each auction and the data related to completed auctions. The *User DB* contains the table `UserData`, storing data on the users, such as their username, password, role, etc.

[1] Not all tests work yet.

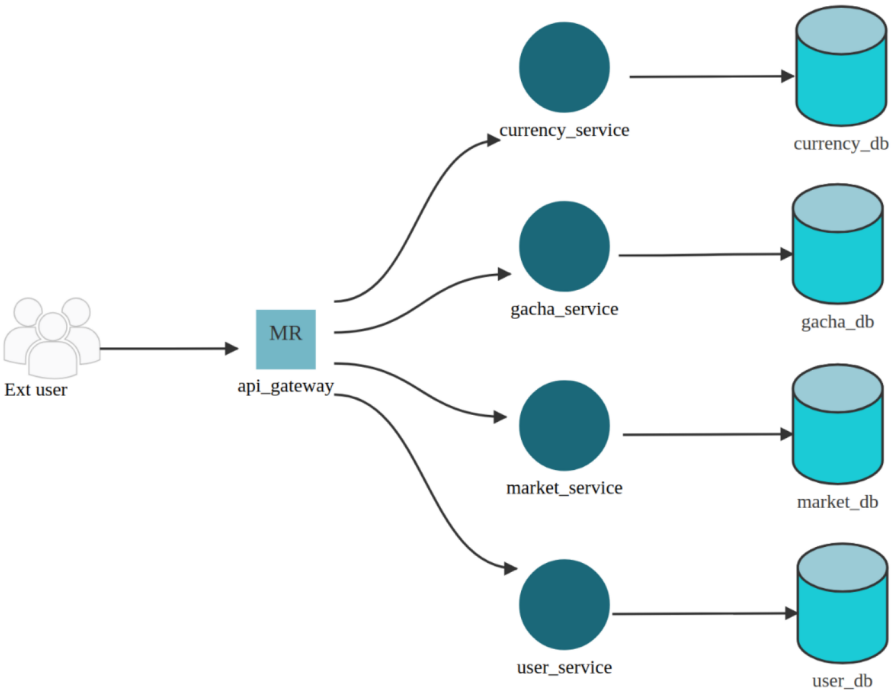

Figure 2: Gachana architecture represented in MicroFreshener.

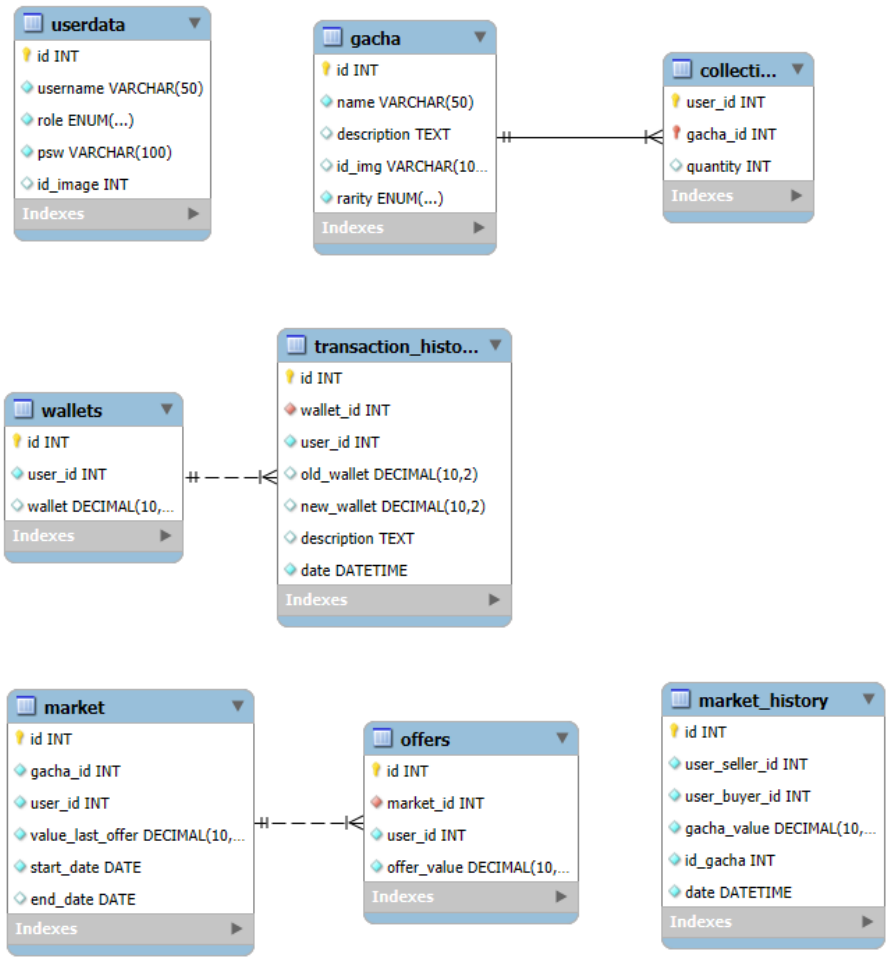

Figure 3: Database architecture

2 Flow of player operations

The players are the main users of the Gachana application and they are allowed to perform some operations, such as checking the gachas they own. This section will show the flow within the backend of the application of four interesting player operations.

First flow: creation of a player account

If a new user wants to create an account to become a player in Gachana application, they will need to send a `POST` request to the API Gateway, which will have the following structure:

`http://localhost/user/signup`

Given that the user does not already have an account, this request will not contain an authorization token, but in the body there will be the Json fields `username` and `psw`. The request is passed by the API Gateway to the User microservice, which can be found on `http://user_service:5000/`. The User microservice will get he request body and check that the necessary fields `username` and `psw` are present. In case the body does not contain them, it will respond with an error message along with code 400. Otherwise, it will establish a connection with the User DB and it will check whether there is any user with the same value of username: if there is, it will return an error message along with the code 400. If the username is unique, it will proceed inserting the new user into the User DB and then returning a message saying the operation was successful along with code 201.

Second flow: visualization of a player's collection

If a player wants to check their collection, i.e. to see the gacha they already own, they will send a `GET` request to the API Gateway with the following structure:

`http://localhost/gacha/collection`

The request header will contain an authorization field, whose value will be an encrypted token used for authentication and authorization. The API gateway will pass the request to the Gacha microservice, reaching it on `http://gacha_service:5001/`. The Gacha microservice will then use the token in order to check that the request comes from a player and to get the player's `user_id`. If the token is missing or the user does not have the authorization to perform this action, the response will be an error message along with the code 403. If the user data could not be accessed through the token, the response will still contain an error message with code 401. If the checks do not return any error, the Gacha microservice will open a connection with the Gacha DB. It will then execute a query to get the gachas owned by the player with id `user_id`. The response of the database will be checked and, if nothing has been found, the Gacha microservice will return a message saying that the player does not own any gachas yet, otherwise it will return the gachas in the collection. In any case, the response code will be 200.

Third flow: buying in-game currency

If a player wants to buy in-game currency, they will need to send a `PATCH` request to the API Gateway with the following structure:

`http://localhost/currency/buy_currency`

The request header will contain an authorization field, which will consist of a token used for authentication and authorization. The API Gateway will pass the request to the Currency microservice, which can be found on `http://currency_service:5003/`. The Currency microservice will use the token to check that the request comes from a player and to get the player's `user_id`. If the token is missing or the user does not have the authorization to perform this action, the response will be an error message along with the code 403. If the checks do not return any error, the Currency microservice will get the request body to get the data. In particular it will perform a check on the amount of real money to exchange and then simulate a payment. If one of these two operation fails, the response will contain an error message along with code 400. In case of success, the Currency microservice will open a connection with the Currency DB. It will then check the wallet information for that `user_id` and try to update both the `Wallets` table and the `Transaction_History` table. In case one of the two operations fails, a rollback is performed and the response will contain an error message along with code 500. Otherwise the Currency microservice will return a message containing the amount added to the user's wallet, along with code 200.

Fourth flow: seeing offers for an auction

If a player wants to see the offers for one of their auctions, they will need to send a `GET` request to the API Gateway with the following structure:

`http://localhost/market/<market_id>/offers`

where `market_id` is the identifier of their auction. The request header will contain an authorization field, whose value will be an encrypted token used for authentication and authorization. The API gateway will pass the request to the Market microservice, reaching it on `http://market_service:5002/`. The Market microservice will use the token to check that the request comes from a player and to get the player's `user_id`. If the token is missing or the user does not have the authorization to perform this action, the response will be an error message along with the code 403. If the user data could not be accessed through the token, the response will still contain an error message with code 401. If the checks do not return any error, the Market microservice will open a connection with the Market DB to get the auction `market_id` corresponding to user `user_id`. If there is no such auction, it will return an error message along with code 404. If the auction is not owned by user `user_id`, it will return an error message along with code 403. Otherwise it will get from `Offer` table all the offers corresponding to auction `market_id` and return them along with code 200.