UNIVERSITÀ DI PISA

ASE 2024/2025
First Prototype Delivery

EzGacha Team

Gioele Dimilta
Andrea Mugnai
Jacopo Tucci

Contents

1 Introduction

TBD...

2 Architecture

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

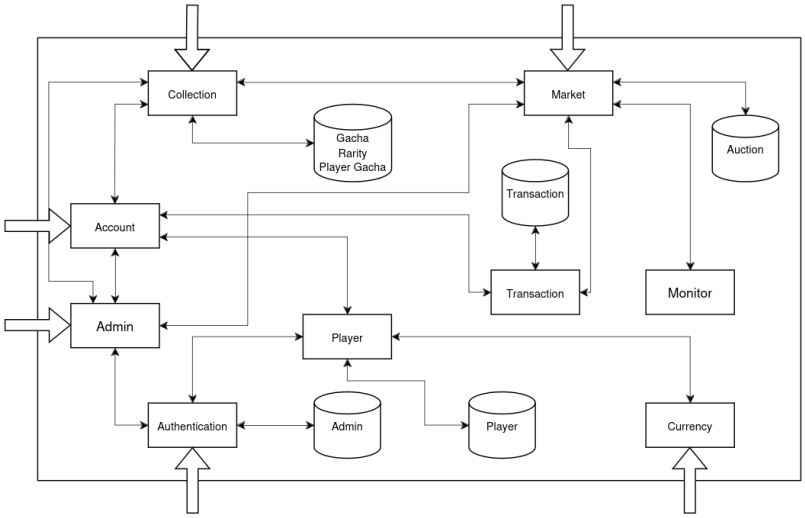

Figure 1: General architecture

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*)

Player Service and *Transaction Service* manage the access to the databases (*Player* and *Transaction* tables). Players and admins can access only the API

exposed by a *CaddyServer* web 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.

2.1 Database

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

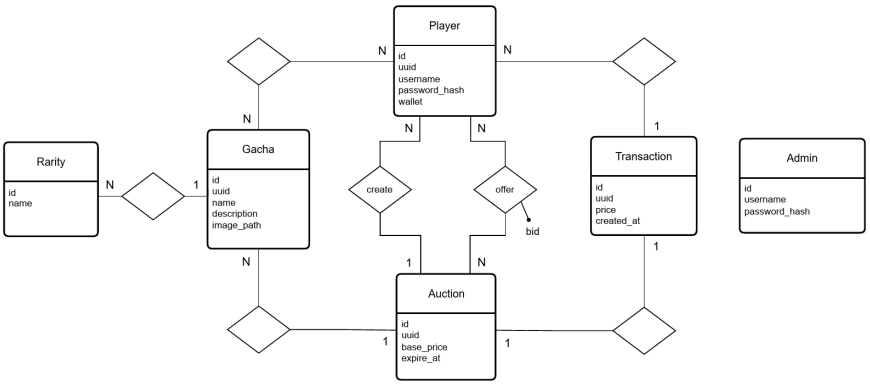

Figure 2: Database structure

The *N-to-N* relationship between *Player* and *Gacha* will be 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, Gacha_Player
Transaction	Transaction
Market	Auction, Bid

2.2 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 *Gacha Service*) and transaction history. In particular, 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 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 when expire time comes. 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 Testing

TBD...

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*. Enter inside the root project directory and run the following commands to build the images and run the containers.

```
cd src
docker compose up -d --build
```