



# **Gacha Collection Beckend**

290AA – Advanced Software Engineering

Group:

Ardizzoni Francesco

Del Castello Diego

Prestifilippo Colombrino Mattia

Tortorelli Felice

# Catalogue – All 151 Pokémon from 1<sup>o</sup> gen

- Common (54,45%)



- Uncommon (40%)



- Rare (5%)



- Epic (0,5%)



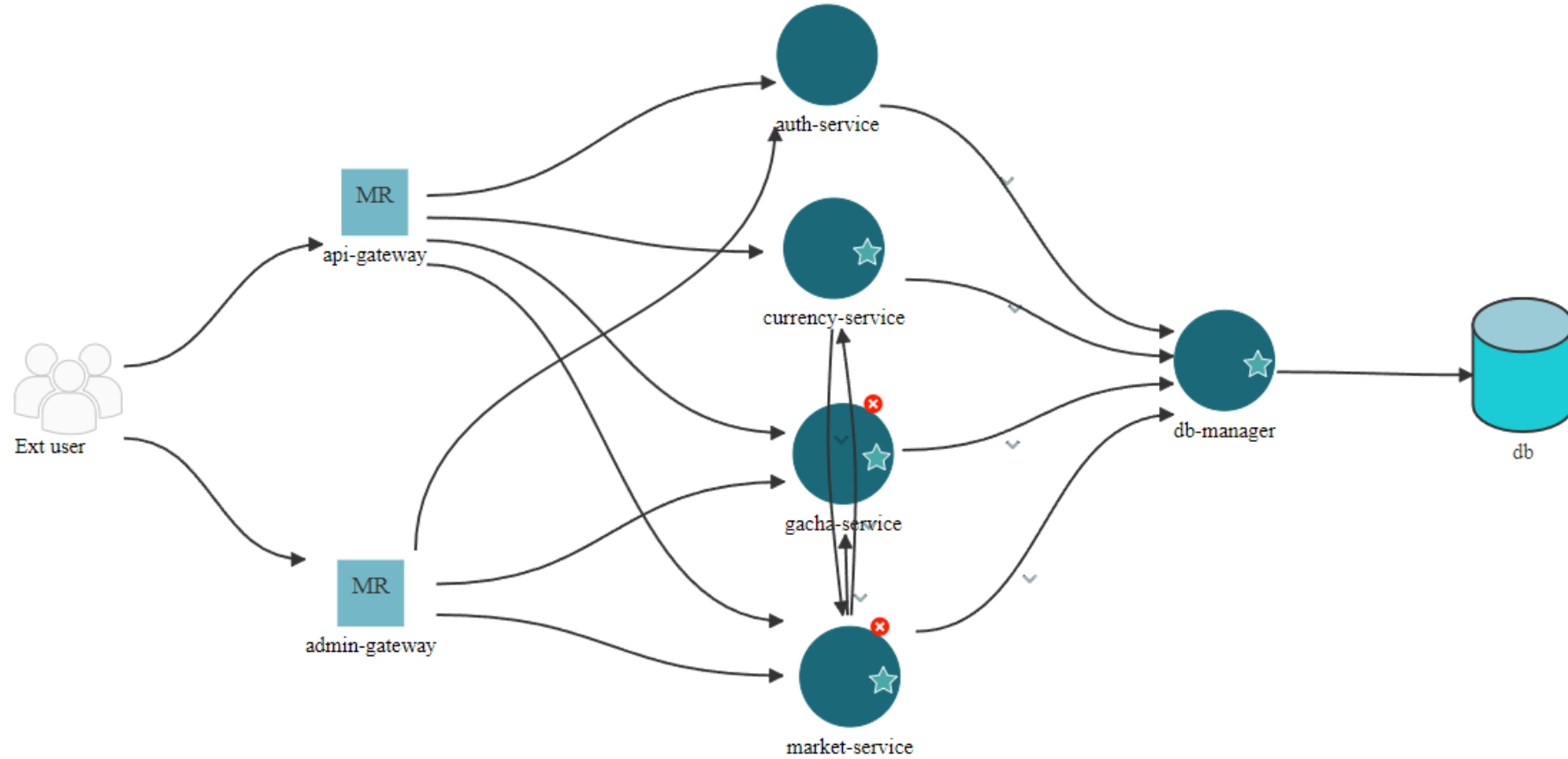
- Legendary (0,05%)



- In-game currency: Pokedollars ₱



# Microfreshener



# The Database

admin	
AdminId	INT
FirstName	VARCHAR(45)
LastName	VARCHAR(45)
Password	VARCHAR(255)
Email	VARCHAR(45)
Salt	VARCHAR(255)
Indexes	

user	
UserId	INT
FirstName	VARCHAR(45)
LastName	VARCHAR(45)
Email	VARCHAR(45)
Password	VARCHAR(255)
CurrencyAmount	INT
Salt	VARCHAR(255)
Indexes	

gacha	
GachaId	INT
Name	VARCHAR(45)
Type1	VARCHAR(45)
Type2	VARCHAR(45)
Total	INT
HP	INT
Attack	INT
Defense	INT
SpAtt	INT
SpDef	INT
Speed	INT
Rarity	VARCHAR(45)
Link	VARCHAR(45)
Indexes	

transaction	
TransactionId	INT
UserOwner	INT
GachaId	INT
RequestingUser	INT
StartingPrice	INT
ActualPrice	INT
EndDate	VARCHAR(45)
SendedTo	INT
Indexes	



# The micro-services

## -api-gateway

This API Gateway acts as the main entry point for client requests, forwarding them to the appropriate microservices for processing. It integrates with multiple services.

## -admin-gateway

This API Gateway acts as the entry point for admin requests, forwarding them to the appropriate microservices for processing. It integrates with multiple services.

## -auth-service

This API is part of a microservices architecture and serves as the user management service for the application. It provides operations such as register a new user or authenticate a user

## -currency-service

This API is part of a microservices architecture and serves as the currency management service for the application. It provides operations for managing in-game currency and rolling gacha items.

## -gacha-service

This API is part of a microservices architecture and serves as the gacha management service for the application. It provides operations for managing gachas and retrieving gacha infos.

## -market-service

This API is part of a microservices architecture and handles transactions related to gacha items and user purchases. It interacts with other services to track and update user transactions.

## -db-manager

This API is part of a microservices architecture and handles all the queries to the database.

# The auth service datapath

The auth-related service for the user is exposed externally to clients on port 8000 via an API gateway. The related service for the admin is exposed to clients on port 8080 via an admin\_gateway. When an HTTPS request is received for a service related to CRUD operations for authentication, the admin/API\_gateway forwards the HTTPS request to the authentication service on port 8001.

The authentication service itself exposes APIs on port 8001 to add, modify, delete update and view an account in the Db.

To interface with the database, the authentication service sends an HTTPS request to the db\_manager, which provides APIs for database interfacing services, as perform sql query, and return the result to the authentication service. These database-related APIs are exposed by the db\_manager on port 8005.

# The currency service datapath

The currency-related service for the user is exposed externally to clients on port 8000 via an API gateway. When an HTTPS request is received for a service related to the operations for currency, the API\_gateway forwards the HTTPS request to the currency service on port 8004.

The currency service itself exposes APIs on port 8004 to roll a gacha and buy in-game currency.

To interface with the database, the currency service sends an HTTPS request to the db\_manager, which provides APIs for database interfacing services, as perform sql query, and return the result to the currency service. These database-related APIs are exposed by the db\_manager on port 8005.

The currency service communicates also with market service to handle operations like creating a new transaction in the market when a roll is done.

# The gacha service datapath

The gacha-related service for the user is exposed externally to clients on port 8000 via an API gateway. The related service for the admin is exposed to clients on port 8080 via an admin\_gateway. When an HTTPS request is received for a service related to CRUD operations for gacha, the admin/API\_gateway forwards the HTTPS request to the gachaservice on port 8002.

The gachaservice itself exposes APIs on port 8002 to add, modify, delete, and view a single gacha, or the entire collection of gachas, or a single or the entire collection of my own gachas.

To interface with the database, the gachaservice sends an HTTPS request to the db\_manager, which provides APIs for database interfacing services, as perform sql query, and return the result to the gacha-service. These database-related APIs are exposed by the db\_manager on port 8005.



# The market service datapath

The market-related service for the user is exposed externally to clients on port 8000 via an API gateway. The related service for the admin is exposed to clients on port 8080 via an admin\_gateway. When an HTTPS request is received for a service related to:

USER:

see the auction market, bid for a gacha from the market, view my transaction history, receive a gacha when I win an auction, set an auction for one of my gacha, receive in-game currency when someone win my auction and receive my in-game currency back when I lost an auction operations for currency

ADMIN:

see the auction market, see a specific auction, modify a specific auction and see the market history.

The admin/API\_gateway forwards the HTTPS request to the market service on port 8003.

The market service communicates also with currency and gacha services to handle operations like creating a new auction or put a bid on an auction.

# The authentication service datapath

User story	Endpoint	Microservices
create my game account/profile	/register /register_admin	admin/api_gateway, auth_service, db_manager
delete my game account/profile	/delete_user /delete_admin	admin/api_gateway, auth_service, db_manager
modify my account/profile	/update_user /update_admin	Admin/api_gateway, auth_service, db_manager
login and logout from the system	/login /login_admin	admin_gateway, auth_service, db_manager
check all users accounts/profiles	/check_users_profile	admin_gateway, auth_service, db_manager
check/modify a specific user account/profile	/update_specific_user	admin_gateway, auth_service, db_manager
check a specific player currency transaction history	/specific_history	admin_gateway, auth_service, db_manager
check a specific player market history	/specific_market_history	admin_gateway, auth_service, db_manager

# The currency service datapath

User story	Endpoint	Microservices
use in-game currency to roll a gacha	/roll	api_gateway, currency_service, market_service, db_manager
buy in-game currency	/buy_currency	api_gateway, currency_service, db_manager
have different level of roll rarity	/golden_roll	api_gateway, currency_service, db_manager

# The gacha service datapath

User story	Endpoint	Microservices
Check all the (system) gacha collection	/gacha/get	admin/api_gateway, gacha_service, db_manager
check a specific (system) gacha	/gacha/get/<int:gacha_id> /gacha/getName/<string:name>	admin/api_gateway, gacha_service, db_manager
modify the gacha collection	/gacha/add	admin_gateway, gacha_service, db_manager
modify a specific gacha information	/gacha/update/<int:gacha_id>	admin_gateway, gacha_service, db_manager
modify the gacha collection	/gacha/delete/<int:gacha_id>	admin_gateway, gacha_service, db_manager
see my gacha collection	/gacha/mygacha	api_gateway, gacha_service, db_manager
/gacha/mygacha/<int:gacha_id>	/gacha/mygacha/<int:gacha_id>	api_gateway, gacha_service, db_manager

# The market service datapath

User story	Endpoint	Microservices
see the auction market	/auction /auction	admin/api_gateway, market_service, db_manager
set an auction for one of my gacha	/new_auction	api_gateway, gacha_service market_service, db_manager
bid for a gacha from the market	/bid/<transaction_id>	api_gateway, market_service, db_manager
view my transaction history	/my_transaction_history	api_gateway, market_service, db_manager
receive a gacha when I win an auction	/bid/<transaction_id>	api_gateway, market_service, db_manager
receive in-game currency when someone win my auction	/bid/<transaction_id>	api_gateway, market_service, db_manager
receive my in-game currency back when I lost an auction	/bid/<transaction_id>	api_gateway, market_service, currency_service, db_manager

# The market service datapath (pt.2)

User story	Endpoint	Microservices
see a specific auction	/auction/<int:transaction_id>	admin_gateway, market_service, db_manager
modify a specific auction	/close_auction/<int:transaction_id>	admin_gateway, market_service, db_manager
see the market history	/auction/history	admin_gateway, market_service, db_manager



# Market rules

The operations that are possible in the market are:

- 1) Roll a gacha;
- 2) Create an auction.

The difference between an auction and a normal roll can be seen looking at the attribute 'UserOwner' because for normal rolls, it is set to null, representing that the UserOwner was the system. When an user create a new auction, he/she is the UserOwner.

TransactionId	UserOwner	GachaId	RequestingUser	StartingPrice	ActualPrice	EndDate	SendedTo	
1	NULL	151	1	10	10	2024-12-05 17:53:16	NULL	roll
2	1	151	NULL	200	200	2024-12-11 23:59:59	NULL	auction

When an user put a bid on an auction, he/she plays the role of RequestingUser.

TransactionId	UserOwner	GachaId	RequestingUser	StartingPrice	ActualPrice	EndDate	SendedTo
1	NULL	151	1	10	10	2024-12-05 17:53:16	NULL
2	1	151	99	200	201	2024-12-11 23:59:59	NULL

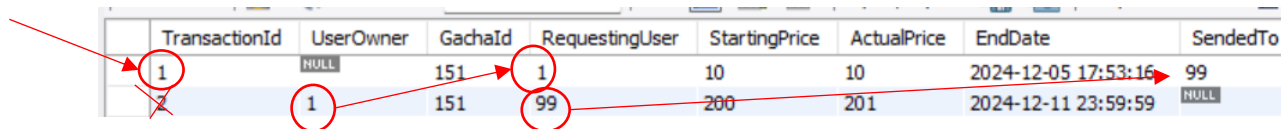
The owner's Currency amount is updated immediately and so for the RequestingUser.

# Market rules

If some user raise on the bid, he becomes the RequestingUser. The old RequestingUser get his bid back.

In order to bid successfully, you need to have sufficient currency amount and you need to put a bid higher than the ActualPrice. You can't raise a bid if the last offer is yours. You can't bid on an auction already closed (EndDate passed). If you bid at the last second of the auction everything is fine.

The last thing that needs to be pointed out is the SentedTo attribute. An admin has the power of close an auction if he wants to. When he does this the SentedTo attribute, in the record where the UserOwner got the gacha, is setted with the RequestingUser ID.



The screenshot shows a database table with the following columns: TransactionId, UserOwner, GachaId, RequestingUser, StartingPrice, ActualPrice, EndDate, and SentedTo. Two rows are visible. The first row has TransactionId 1, UserOwner NULL, GachaId 151, RequestingUser 1, StartingPrice 10, ActualPrice 10, EndDate 2024-12-05 17:53:16, and SentedTo 99. The second row has TransactionId 2, UserOwner 1, GachaId 151, RequestingUser 99, StartingPrice 200, ActualPrice 201, EndDate 2024-12-11 23:59:59, and SentedTo NULL. Red circles highlight the RequestingUser values (1 and 99) and the SentedTo values (99 and NULL). Red arrows indicate the flow of information: from the first row's RequestingUser (1) to the second row's RequestingUser (99), and from the first row's SentedTo (99) to the second row's SentedTo (NULL).

TransactionId	UserOwner	GachaId	RequestingUser	StartingPrice	ActualPrice	EndDate	SentedTo
1	NULL	151	1	10	10	2024-12-05 17:53:16	99
2	1	151	99	200	201	2024-12-11 23:59:59	NULL

# Unit Testing

Every tests were deployed in every microservices in the file *main\_test.py* and executed with the *Dockerfile\_test*.

To ensure an isolated testing environment, an in-memory simulated database (*mock\_database*) was used.

The **mock database** is implemented as a dictionary and simulates the user, gacha, and transaction tables. These mock tests intercept all calls made to the db-manager (and other services), simulating their operation.

Every operation was tracked using the *mock\_save\_last()* function, which records the operation performed, the parameters passed, and the resulting output.

# Integration Testing

Designed to verify the proper functioning of various services of the application in a combined environment, ensuring that all services work correctly together.

Aim to ensure that communication between the various microservices operates smoothly.

Use of **docker-compose** to test the entire architecture in isolation.

In the case of registration, the user registers using the /register endpoint of the authentication service and obtains a JWT token. Token is used to allow a user for applying for an operation

Test made on the API-gateway for all the microservices

# Load Testing

To verify the application's performance, Locust is used to ensure the system can handle a high number of simultaneous requests without performance degradation.

After registration, each user attempts to log in via the */login* endpoint. If successful, the JWT token is stored for subsequent operations.

A series of CRUD operations are performed on Gacha objects.

Allows for verifying the capacity of the authentication service to handle a large number of requests.

# Security - data

## Data Encrypted:

- 1) **Password Hash:** Represents the securely hashed version of the user's password.
- 2) **Salt:** A unique random string used to add variability to the password hashing process to prevent precomputed attacks (e.g., rainbow tables).

## Database Storing Them:

- The user and admin tables in the db\_manager service store the password hash and the salt for each user.

## Encryption and Decryption Locations:

### 1) Encryption:

- Performed in the auth\_service during the registration using the hash\_password function.
- The password is hashed using 'bcrypt' and a generated salt, which is then Base64-encoded to ensure safe storage and transfer.

### 2) Decryption:

- Strictly speaking, passwords are not decrypted. Instead, during login, the user-provided password is hashed with the stored salt and compared against the stored hash using the verify\_password function in the auth\_service.

This approach ensures secure handling of sensitive information across microservices and protects the data at rest in the database.



# Security–Authentication and Authorization

## Scenario Selected: Centralized Authentication and Authorization

In this system, the **utilis** acts as a **centralized authority** for authentication and token generation. The tokens are signed and validated exclusively by the utilis, ensuring consistent and secure handling of access control.

### Steps to Validate a Token:

#### 1. Token Issuance:

- The **utilis** generates a JWT during login or registration.
- The token is signed using the secret key stored securely in the **secret\_key.env** file.
- The token contains the user's claims (payload) and an expiration time.

#### 2. Token Validation:

Each microservice requiring authorization:

- Retrieves the token from the Authorization header.
- Sends the token to the **utilis** for validation.

#### 3. The **auth\_service**:

- Decodes the token using the same secret key.
- Checks the payload claims (e.g., expiration, role).
- Verifies the token against a blacklist to ensure it has not been invalidated (e.g., after logout).

# Security–Authentication and Authorization

## Access Token Payload Format:

```
payload = {  
    "sub": user_id,  
    "role": role,  
    "pass": user_pass,  
    "iat": datetime.datetime.now(datetime.timezone.utc),  
    "exp": datetime.datetime.now(datetime.timezone.utc) + datetime.timedelta(seconds=JWT_EXPIRATION_TIME)  
}
```

## Validation Flow:

### Token Issuance:

1. **Input:** Email, password → **Output:** JWT (signed with secret key)

### 1.Token Validation:

1. **Input:** JWT → **Validation:** Decode → Verify payload → Check blacklist

### 2.Access Granted or Denied:

1. If the token is valid, access is granted.
2. If invalid, a 401 Unauthorized error is returned.

# Security – Analyses

Bandit's final table:

```
-----  
Code scanned:  
  Total lines of code: 2831  
  Total lines skipped (#nosec): 0  
  Total potential issues skipped due to specifically being disabled (e.g., #nosec BXXX): 0  
  
Run metrics:  
  Total issues (by severity):  
    Undefined: 0  
    Low: 3  
    Medium: 102  
    High: 76  
  Total issues (by confidence):  
    Undefined: 0  
    Low: 95  
    Medium: 8  
    High: 78  
Files skipped (0):
```

# Security – Analyses

Pip-audit final table:

```
PS C:\Users\faloo\OneDrive\Desktop\UNI PISA\2 anno 1 semestre\Advanced Software Engineering\Progetto\TBAs\src> pip-audit --vulnerability-service osv
No known vulnerabilities found
```

# Additional features

User story	Endpoint	Microservices
have different level of roll rarity	/golden_roll	api_gateway, currency_service, db_manager
check all users accounts/profiles	/check_users_profile	admin_gateway, auth_service, db_manager
check/modify a specific user account/profile	/update_specific_user	admin_gateway, auth_service, db_manager
check a specific player currency transaction history	/specific_history	admin_gateway, auth_service, db_manager
check a specific player market history	/specific_market_history	admin_gateway, auth_service, db_manager
modify the gacha collection	/gacha/add	admin_gateway, gacha_service, db_manager
modify a specific gacha information	/gacha/update/<int:gacha_id>	admin_gateway, gacha_service, db_manager
modify the gacha collection	/gacha/delete/<int:gacha_id>	admin_gateway, gacha_service, db_manager
see a specific auction	/auction/<int:transaction_id>	admin_gateway, market_service, db_manager
modify a specific auction	/close_auction/<int:transaction_id>	admin_gateway, market_service, db_manager
see the market history	/auction/history	admin_gateway, market_service, db_manager