

G.L.S. group

Giuseppe Pio La Tosa

Giuseppe Scarabaggio

Simone Setzu

Sommario

- Gachas overview..... 3
- Architecture..... 4
- User Stories 6
- Market rules..... 10
- Testing..... 12
- Security – Data..... 14
- Security – Authorization and Authentication 15
- Security – Analyses 18
- Additional Features 21

Gachas overview

In our project, Pokémon are featured as gachas, each characterized by their **name, image, and rarity**. Every type of Pokémon is identified within the system by a unique `item_id`, representing its specific characteristics. However, when a user rolls for a Pokémon, they can obtain multiple instances of the same Pokémon. These instances, while sharing the same `item_id`, are distinguished from each other by a unique `instance_id`, allowing the system to treat them as separate entities. For example, a user who already owns a Pikachu can obtain another Pikachu with identical attributes, but the two will be differentiated by their unique `instance_id`.



Rarity Management and Drop Probability in the Gacha System

In the system, each gacha belongs to a rarity class (*common*, *rare*, etc.) that determines the likelihood of obtaining it during a roll. Specifically:

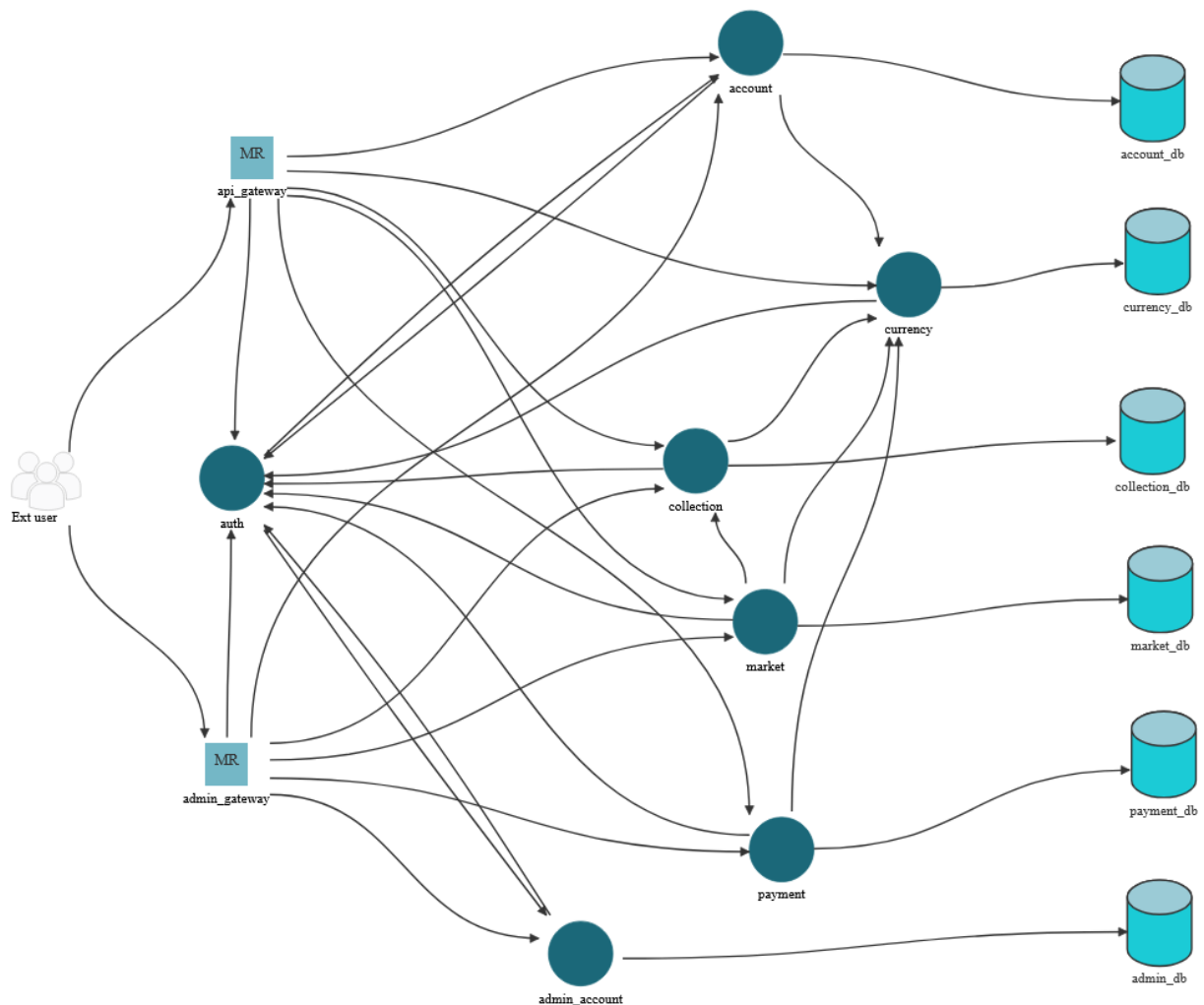
- The percentages associated with each rarity class, as shown in the image above, indicate the overall probability of obtaining any gacha belonging to that class.
- The probability of each individual gacha is calculated based on its rarity class and the total number of gachas in that class, using the formula:

$$ProbItem = \frac{Rarity\ class\ probability}{\#gacha\ belonging\ to\ the\ rarity\ class}$$

This mechanism ensures that all gachas within the same rarity class have evenly distributed probabilities, maintaining a balanced system.

Thanks to this approach, the system can automatically adapt when an administrator adds or removes a gacha from the global collection (only possible if it is not owned by any user), ensuring that the probabilities remain balanced and consistent, regardless of the number of gachas in each class.

Architecture



Here's a brief description of each microservice and their interactions:

1. API Gateway:

- o Acts as a single entry point for external users to interact with the microservices.
- o Routes requests to account, collection, market, currency, auth and payment for user API.

2. Admin API Gateway:

- o Acts as a single entry point for external admins to interact with the microservices.
- o Routes requests to account, admin_account, collection, market, auth and payment for admin API.

3. Auth:

- o Acts as authorization server in distributed authentication scenario.
- o Every microservice (excepted for the gateway and admin gateway) when started send a request to auth to obtain the public key for token validation
- o Interacts with account and admin_account for credential validation

4. Account:

- o Manages user accounts:

- 1) users can create, delete, modify e obtain their account

- 2) admins can view data about all user accounts and change user email for a specific user

- o Interacts with currency for initialize user amount after user creation

5. Admin Account:

- o Manage admins accounts including creation and deletion.

6. Currency:

- o Handles operations related to rubies: add, sub and get rubies amount.

- o The only request exposed for API gateway allows users to check their rubies amount.

7. Collection:

- o Handles operations related to collection of gachas:

- 1) users can obtain all type of system gachas, see the specific gacha by its item_id, see his collection and a specific item of his collection by its instance_id and roll gachas to add new items to his collection.

- 2) admins can also see all type of system gachas and a specific gacha by its item_id, modify and delete an existing gacha and add a new gacha in the system.

- o Interacts with currency in order to pay the roll operation.

8. Market:

- o Manages marketplace functionalities:

- 1) users can get all open auctions in the market, buy a gacha bidding to a specific auction, sell one of his gacha creating an auction and see his transactions history

- 2) admins can see all open auctions in the market, see the transactions history of all users or of one specific user, get and close a specific auction.

- o Interacts with currency for auction handling and with collection to check if a user own the gacha that he wants to sell and to reassign it from the seller to the buyer.

9. Payment:

- o Handles payment transactions:

- 1) users can buy new rubies in order to increase their rubies amount.

- 2) admins can get all users' currency_history.

- o Interacts with currency to add equivalent rubies in case of successful transaction

We added our architecture to **MicroFreshner** for analysis, and the tool identified a **Wobbly Service Interaction** smell. This issue was caused by unstable or improper communication patterns between services. To resolve the problem, we utilized the **register_error** mechanism provided by Flask, which improved error handling and stabilized the interactions between the services. This adjustment eliminated the identified smell and enhanced the robustness of our architecture.

User Stories

PLAYER

| INDEX | METHOD | ENDPOINT/NOTES | COMPONENTS |
|-------|--------|---|---|
| 4 | POST | /user | api_gateway, account, currency, account_db |
| 5 | DELETE | /user/{user_id} | api_gateway, account, account_db |
| 6 | POST | /user/{user_id} | api_gateway, account, account_db |
| 7 | POST | /user/auth | api_gateway, auth, account, account_db |
| 7 | DELETE | /user/{user_id}/auth | api_gateway, auth |
| 8 | POST | /user/{user_id}/auth Fulfilled by adding HTTPS for credentials confidentiality in transit and an attempt limiter in login requests to avoid brute force attacks. | api_gateway, auth, account, account_db |
| 9 | GET | /user/{user_id}/mycollection | api_gateway, collection, collection_db |
| 10 | GET | /user/{user_id}/instance/{instance_id} | api_gateway, collection, collection_db |
| 11 | GET | /user/{user_id}/collection | api_gateway, collection, collection_db |
| 12 | GET | /user/{user_id}/item/{item_id} | api_gateway, collection, collection_db |
| 13 | PUT | /user/{user_id}/roll | api_gateway, collection, currency, collection_db, currency_db |
| 14 | POST | /user/{user_id}/payment | api_gateway, payment, payment_db |

| | | | |
|----|-----|---|---|
| 15 | | All requests involving the user's in-game currency are authorized via JWT token by verifying the user's identity. | |
| 16 | GET | /user/{user_id}/market_list | api_gateway, market, market_db |
| 17 | PUT | /user/{user_id}/instance/{instance_id}/auction | api_gateway, market, collection, collection_db, market_db |
| 18 | PUT | /user/{user_id}/market/{market_id}/bid | api_gateway, market, currency, currency_db, market_db |
| 19 | GET | /user/{user_id}/transactions_history | api_gateway, market, market_db |
| 20 | | Handled by periodic task in market (check_expired_auctions) | market, collection, collection_db |
| 21 | | Handled by periodic task in market (check_expired_auctions) | market, currency, currency_db |
| 22 | PUT | /user/{user_id}/market/{market_id}/bid | api_gateway, market, currency, currency_db, market_db |
| 23 | PUT | /user/{user_id}/market/{market_id}/bid | api_gateway, market, currency, currency_db, market_db |

ADMIN

| INDEX | METHOD | ENDPOINT | COMPONENTS |
|--------------------|--------|-------------------|--|
| Additional feature | POST | /admin | admin_api_gateway, admin account, admin_account_db |
| Additional feature | DELETE | /admin/{admin_id} | admin_api_gateway, auth, admin account, admin_account_db |

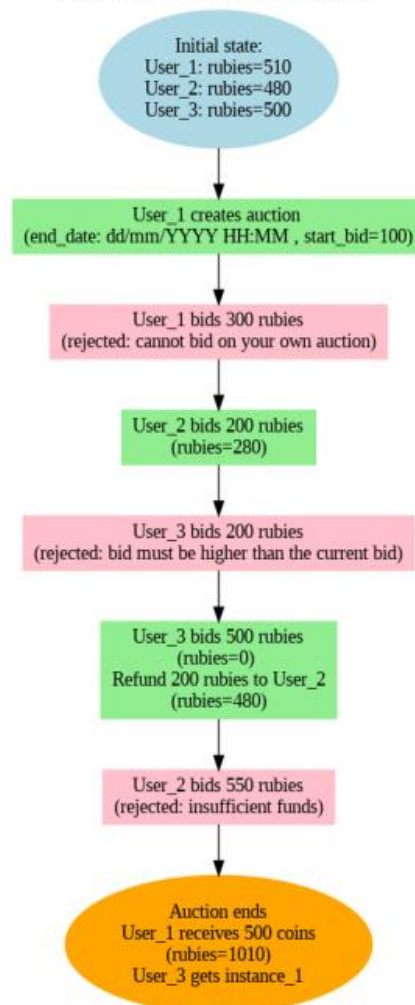
| | | | |
|----------------------------|--------|---|--|
| 4 | POST | /admin/auth | admin_api_gateway, auth, admin account, admin_account_db |
| 4 | DELETE | /admin/{admin_id}/auth | admin_api_gateway, auth |
| 5 Additional feature | GET | /admin/{admin_id}/users | admin_api_gateway, account, account_db |
| 6 Additional feature | GET | /admin/{admin_id}/user/{user_id} | admin_api_gateway, account, account_db |
| 6 Additional feature | POST | /admin/{admin_id}/user/{user_id} | admin_api_gateway, account, account_db |
| 7 Additional feature | GET | /admin/{admin_id}/user/ {user_id}/currency_history | admin_api_gateway, payment, payment_db |
| 8 Additional feature | GET | /admin/{admin_id}/user/{user_id}/trans actions_history | admin_api_gateway, market, market_db |
| 9 | GET | /admin/{admin_id }/collection | admin_api_gateway, collection, collection_db |
| 10 | PUT | /admin/{admin_id}/item | admin_api_gateway, collection, collection_db |
| 10 | DELETE | /admin/{admin_id}/item/{item_id} | admin_api_gateway, collection, collection_db |
| 11 | GET | /admin/{admin_id}/item/{item_id} | admin_api_gateway, collection, collection_db |
| 12 | POST | /admin/{admin_id}/item/{item_id} | admin_api_gateway, collection, collection_db |

| | | | |
|-----------------------------|------|--|---|
| 13 Additional feature | GET | /admin/{admin_id}/market_list | admin_api_gateway, market, market_db |
| 14 Additional feature | GET | /admin/{admin_id}/market/{market_id} | admin_api_gateway, market, market_db |
| 15 Additional feature | POST | /admin/{admin_id}/market/{market_id} | admin_api_gateway, market, market_db |
| 16 Additional feature | GET | /admin/{admin_id}/transactions_history | admin_api_gateway, market, market_db |

Market rules

1. A user can put a gacha from their collection up for auction, provided it is not already in an open auction (endpoint /user/{user_id}/instance/{instance_id}/auction), by setting a starting price (start_bid) and an end date for the auction (which can be any date in the future).
2. Once a user has successfully started an auction, other users can view it in the list of open auctions (endpoint /user/{user_id}/market_list). A user can participate in the auction by placing a bid higher than the current one (endpoint /user/{user_id}/market/{market_id}/bid). If the operation is successful (it fails if the user does not have enough rubies), the bid amount of rubies is deducted from the user's credit, and the bid cannot be canceled. If another user outbids them, the previous bidder's rubies are refunded. Additionally, the same user is allowed to place consecutive bids to increase the bid amount.
3. The admin is able to close the auction before it expires. If a user has placed a bid, the bidder's rubies are refunded.
4. When the auction expires the gacha is assigned to the winner, and the seller is credited with the coins. If there are no bids in an auction the gacha remains with the seller.

Auction Flowchart



Additional considerations

Auction Expiration Management

To manage the expiration of auctions, we have implemented an automated system that **periodically monitors** active auctions (e.g every 10 seconds, as defined in the configuration), checking if they have expired based on their end date. When an auction is found to be expired, the system automatically updates its status and handles subsequent actions, such as transferring items between users and updating the currencies.

Additionally, to prevent a user from placing a bid on an expired auction, we compare the bid date with the auction's expiration date. This ensures that bids can only be placed while the auction is still active, avoiding incorrect transactions and maintaining the integrity of the system.

Data Inconsistency Risks

The management of bids in auctions can lead to data inconsistency issues due to the fact that the user's balance and the auctions are stored in two separate databases, interacting with two different microservices. For example:

1. **Error in refunding the previous bidder:** If the microservice responsible for refunding the previous bidder malfunctions, the user may not receive the balance of the previous bid, causing inconsistencies in the balances.
2. **Error during auction update:** If the auction database does not update correctly after the bid is recorded, the auction data may become inconsistent, even though the user's balance was incorrectly updated.

In such cases, the application administrator can retrieve the operation logs to identify errors and refund affected users, restoring the correct balance states.

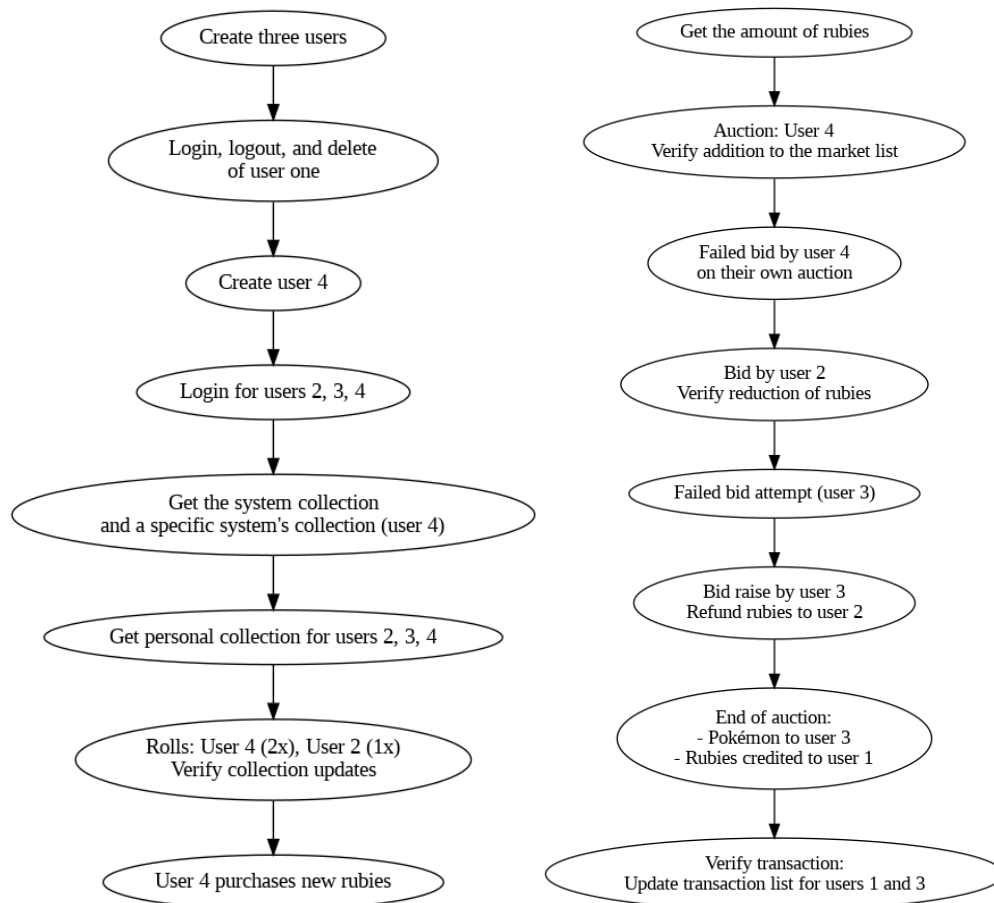
Testing

To ensure that the code has been implemented correctly and meets the required functionality, we developed a series of tests using Postman. We performed two types of tests: unit tests, where we tested each microservice in isolation, and integration tests.

We verified both successful execution cases and failure scenarios to ensure comprehensive coverage of possible outcomes and to validate the robustness of the microservices.

Integration Testing

To test the entire application, we created tests targeting the API Gateway and the Admin API Gateway, developing a scenario that simulated a typical interaction involved users and administrators with our application. These tests covered all operations that users and administrators can perform, verifying that everything works as expected. Below is the flow of the scenario of the user:



Unit Test

For each microservice, we adopted an isolated testing approach by creating a dedicated **docker-compose.yml** file for each component. This file launches the specific microservice along with its database, enabling each component to be tested separately in a controlled environment. This configuration simplified test management, reducing external interactions and allowing exclusive focus on the microservice's logic under test.

Test Environment Configuration with `FLASK_ENV=testing`

In the test configuration process, a key element was the use of the `FLASK_ENV=testing` environment variable, which had two fundamental effects:

1. Mocking Activation:

When a microservice needed to communicate with another service we configured the microservice to operate in testing mode. In this mode, instead of making actual requests to external services, the microservice responded with predefined data through the mocking system, allowing the simulation of responses from other microservices.

2. Token Validation Bypass:

Another significant advantage of this configuration was the bypass of JWT token validation. Specifically, during testing, when the microservice required a token to authenticate a request, the token validation check was skipped. This significantly simplified the execution of unit tests by avoiding the complexity of generating and managing tokens, allowing the tests to focus on the microservice's logic.

Performance Testing with Locust

In addition to unit and integration testing, we conducted performance testing to evaluate the system's behavior under high-load conditions. This was achieved using **Locust**, targeting only the **gateway** of the system. The performance tests specifically focused on the **player endpoints** associated with gacha operations.

Objectives of the Performance Tests:

1. Validate Endpoint Stability under Load

By simulating a high volume of requests, we tested the robustness and responsiveness of the gacha-related endpoints. This included actions like rolling for items, retrieving collections, and accessing item details.

2. Verify Rarity Distribution

A key aspect of these tests was to ensure that the **rarity distribution** of gacha rolls remained accurate and consistent even under significant load. The system was monitored to confirm that the statistical distribution of outcomes adhered to the expected probabilities.

Security – Data

Input Sanitization

To protect our microservices from malicious inputs and potential attacks, we apply various sanitization measures to user inputs and data. Below are the key areas where we enforce sanitization:

1. Text Inputs

- **Email:** We validate emails using regular expressions to ensure they follow the correct format.
- **Text Fields:** All user inputs (e.g., names, messages) are sanitized by escaping potentially dangerous characters (such as <, >, &) to prevent Cross-Site Scripting (XSS) and HTML injection attacks. We achieve this using Python's `escape()` function.

2. SQL Queries (e.g., Database Inputs)

- **SQLAlchemy ORM:** Instead of using direct parameterized queries, we handle all database interactions through SQLAlchemy ORM. This ensures that user inputs are automatically sanitized and escaped, preventing **SQL injection** risks. The ORM layer automatically constructs queries with proper escaping mechanisms, ensuring secure data handling.

3. URL Parameters (e.g., User IDs)

- We validate URL parameters, such as `user_id`, to ensure they adhere to expected formats (e.g., integers for numeric IDs). This prevents attackers from tampering with URLs to perform unauthorized actions.

Example in Flask route: `@app.route('/user/<int:user_id>')`

Data Encrypted at Rest

List of Data Encrypted at Rest:

1. User Passwords:

- **What They Represent:** These are the passwords used by users for authentication within our system.
- **Database:** Stored in the Account Database (`account_db`).
- **Where Encrypted:** We hash the passwords using **SHA256** algorithm (with **PBKDF2 - Password-Based Key Derivation Function 2**) before storing them in the database.
- **Process:**
 - When a user creates an account or changes their password, we hash the password using **PBKDF2:SHA256**.
 - The hashed password is then securely stored in the Account Database.
 - During login, we hash the entered password and compare it with the stored hash to authenticate the user.

2. Transaction Data:

- **What They Represent:** This includes transaction details, such as the amount, status, and related metadata.
- **Database:** Stored in the Payment Database (payment_db).
- **Where Encrypted:** We encrypt transaction data using **AES (AES-256)** before storing it in the database.
- **Process:**
 - When a transaction is completed, we encrypt the transaction data.
 - The encrypted data is stored in the Payment Database.
 - The data is decrypted when needed, such as for generating reports or troubleshooting.

Security – Authorization and Authentication

Our scenario is **distributed** so it means that microservices are capable of validating tokens independently. They do this by using a public key exposed by the **Auth** microservice, which allows them to verify the signature of the JWT tokens issued during login.

1. Key Generation and Setup

1. Generating the Key Pair (Private/Public):

- **Auth** generates a pair of RSA keys:
 - **Private key** for signing JWTs.
 - **Public key** for validating JWTs.

2. Expose the Public Key:

- **Auth** exposes its **public key** through a JWKS endpoint (/well-known/jwks.json). This allows other microservices to retrieve and use this public key to validate incoming JWT tokens.

2. Authentication Process (Token Generation)

1. Login Request:

- A request with user credentials (username, password) is sent to **Auth**.
- Example request: POST /user/auth with JSON data:
- { "username": "user", "password": "password123", "grant_type": "password" }

2. Credential Validation:

- **Auth** call Account/Admin_account for credential validation and if it success, proceeds to the next step.

- We have implemented a login attempt limiter mechanism in **account** and **admin_account** to prevent brute force attacks on the password. The number of login attempts for a user is saved in *Redis*, and when it exceeds a certain number defined in the configuration (e.g 5), the login will return a 400 error until the Redis entry expires (e.g 10 minutes, as defined in the configuration).

3. JWT Generation:

- **Auth** generates a JWT token by:
 - Signing the token with its **private RSA key**.
 - Adding all relevant claims.

4. Respond with JWT:

- **Auth** sends the JWT token back as the response for further requests.

3. Token Validation in Microservices

1. Receiving a Token:

- A microservice receives a request, and the request contains the JWT token in the Authorization header:
- Authorization: Bearer <JWT_TOKEN>

2. Decode and Validate the Token:

- The **microservice** extracts the JWT token and validates it using the **public key** it has previously fetched from **Auth**.
- The validation includes:
 - **Signature validation** using the public key.
 - **Claim validation:** check if the iss (issuer), aud (audience), and exp (expiration) claims are valid.

3. Authorization:

- If the token is valid, the microservice processes the request.
- If the token is invalid, expired, or the signature is incorrect, the request return a **401 Unauthorized** response.

4. Access Control

For endpoints that require it, the microservice checks whether the user can access a specific resource by comparing the **sub** claim from the JWT (which represents user_id or admin_id) with the user_id or admin_id present in the request URL (e.g /user/{user_id}/...):

- If the **sub** claim in the token matches the **user_id/admin_id** in the URL, the request is allowed.
- If the **sub** claim does not match the **user_id/admin_id**, the request return a **403 Forbidden** response.

JWT Access Token Payload (Example)

Here's an example of the **JWT token payload** that would be generated by **Auth**:

| Claim | Value |
|------------------|-------------------------------------|
| iss (Issuer) | localhost |
| sub (Subject) | {user_id}/{admin_id} |
| aud (Audience) | localhost |
| iat (Issued At) | 2024-12-03T10:00:00Z |
| exp (Expiration) | 2024-12-03T11:00:00Z |
| scope (Role) | user/admin |
| jti (JWT ID) | aabbccdd-eeff-1234-5678-91011121314 |

- **iss**: The entity that issued the token.
- **sub**: The user ID or the subject of the token (e.g., 12345).
- **aud**: The intended audience for the token.
- **iat**: The timestamp when the token was issued.
- **exp**: The timestamp when the token will expire.
- **scope**: The role or scope of the user.
- **jti**: A unique identifier for the token to prevent replay attacks.

Additional Considerations: Token Revocation and Logout

Our system **does not** support token revocation. Since JWTs are stateless and the authentication is distributed, logging out **does not** immediately invalidate the token. The token remains valid until it expires.

If a revocation mechanism were to be implemented, **refresh tokens** would need to be used. In this case, the refresh token should be revoked at the time of logout to prevent the issuance of new JWTs, thereby ensuring more secure session management.

Security – Analyses

BANDIT

We have performed a static code analysis using **Bandit** to identify security vulnerabilities and potential issues related to security in our software. In order to do this we have executed **bandit -r .** in src folder and we have obtained these results:

```
Code scanned:
  Total lines of code: 1778
  Total lines skipped (#nosec): 0

Run metrics:
  Total issues (by severity):
    Undefined: 0
    Low: 1
    Medium: 0
    High: 4
  Total issues (by confidence):
    Undefined: 0
    Low: 0
    Medium: 0
    High: 5
Files skipped (0):
```

Low Issues

```
>> Issue: [B311:blacklist] Standard pseudo-random generators are not suitable for security/cryptographic purposes.
  Severity: Low   Confidence: High
  CWE: CWE-330 (https://cwe.mitre.org/data/definitions/330.html)
  More Info: https://bandit.readthedocs.io/en/1.8.0/blacklists/blacklist\_calls.html#b311-random
  Location: .\collection\app\api\user.py:138:18
137     items, probabilities = zip(*item_probabilities)
138     rolled_item = random.choices(items, probabilities, k=1)[0]
139     response = CurrencyHelper().sub_amount(user_id, current_app.config["roll_price"])
```

Bandit's message highlights the use of a pseudo-random generator (`random.choices`) that is not suitable for cryptographic or security purposes. However, in this specific case, the use of the pseudo-random generator is not related to security or cryptography, but rather to the mechanics of object extraction in the gacha system. For this reason, we have decided to disregard this issue.

High Issues

```
>> Issue: [B501:request_with_no_cert_validation] Call to requests with verify=False disabling SSL certificate checks, security issue.
  Severity: High   Confidence: High
  CWE: CWE-295 (https://cwe.mitre.org/data/definitions/295.html)
  More Info: https://bandit.readthedocs.io/en/1.8.0/plugins/b501\_request\_with\_no\_cert\_validation.html
  Location: .\utils\src\utils_helpers\http_client.py:8:15
7         def get(url, **kwargs):
8             return requests.get(url, verify=False, timeout=1, **kwargs)
9
```

Bandit's message indicates that the `verify=False` option disables the validation of SSL/TLS certificates, which can pose a security risk in production applications. However, in this specific case, its use is justified because our application uses self-signed certificates for testing and development, which makes it impossible to verify the certificate using a traditional CA (Certificate Authority). Enabling `verify=False` is necessary to allow communication between systems without SSL errors. For this reason, we have decided to disregard this issue.

PIP AUDIT

We also used pip-audit to perform an additional vulnerability analysis on the Python packages. The commands executed were as follows, repeated for each container:

1. **Access the Docker container:** `docker exec -it container_name /bin/bash`
2. **Install pip-audit:** `pip install pip-audit`
3. **Run pip-audit to analyze vulnerabilities:** `pip-audit`
4. **Automatically apply available fixes:** `pip-audit --fix`

During the analysis, the only identified vulnerability was related to the version of **pyjwt** (2.10.0) in use. This vulnerability was resolved by upgrading the package to version **2.10.1** in requirements.txt files.

```
root@96f093d4d5f1:/app# pip-audit
Found 1 known vulnerability in 1 package
Name Version ID Fix Versions
-----
pyjwt 2.10.0 GHSA-75c5-xw7c-p5pm 2.10.1
root@96f093d4d5f1:/app# pip-audit --fix
Found 1 known vulnerability in 1 package and fixed 1 vulnerability in 1 package
Name Version ID Fix Versions Applied Fix
-----
pyjwt 2.10.0 GHSA-75c5-xw7c-p5pm 2.10.1 Successfully upgraded pyjwt (2.10.0 => 2.10.1)
```

Additionally, this action allowed us to address the alerts identified by **Dependabot**.

Dependabot alerts

[Give feedback](#)[Configure](#)

☒ Clear current search query, filters, and sorts

| <input type="checkbox"/> | <input checked="" type="checkbox"/> 0 Open | <input checked="" type="checkbox"/> 7 Closed | Closed as ▾ | Package ▾ | Ecosystem ▾ | Manifest ▾ | Severity ▾ | Sort ▾ |
|--------------------------|--|---|---|-----------|-------------|------------|------------|--------|
| <input type="checkbox"/> | <input checked="" type="checkbox"/> | PyJWT Issuer field partial matches allowed Low | #7 closed as fixed 3 hours ago • Detected in PyJWT (pip) • src/payment/requirements.txt | | | | | |
| <input type="checkbox"/> | <input checked="" type="checkbox"/> | PyJWT Issuer field partial matches allowed Low | #6 closed as fixed 3 hours ago • Detected in PyJWT (pip) • src/market/requirements.txt | | | | | |
| <input type="checkbox"/> | <input checked="" type="checkbox"/> | PyJWT Issuer field partial matches allowed Low | #5 closed as fixed 3 hours ago • Detected in PyJWT (pip) • src/currency/requirements.txt | | | | | |
| <input type="checkbox"/> | <input checked="" type="checkbox"/> | PyJWT Issuer field partial matches allowed Low | #4 closed as fixed 3 hours ago • Detected in PyJWT (pip) • src/collection/requirements.txt | | | | | |
| <input type="checkbox"/> | <input checked="" type="checkbox"/> | PyJWT Issuer field partial matches allowed Low | #3 closed as fixed 3 hours ago • Detected in PyJWT (pip) • src/auth/requirements.txt | | | | | |
| <input type="checkbox"/> | <input checked="" type="checkbox"/> | PyJWT Issuer field partial matches allowed Low | #2 closed as fixed 3 hours ago • Detected in PyJWT (pip) • src/admin_account/requirements.txt | | | | | |
| <input type="checkbox"/> | <input checked="" type="checkbox"/> | PyJWT Issuer field partial matches allowed Low | #1 closed as fixed 3 hours ago • Detected in PyJWT (pip) • src/account/requirements.txt | | | | | |

DOCKER SCOUT

We also utilized **Docker Scout** to analyze and enhance the security of our Docker images. The process was carried out for the following repository and images:

Repository:

giuseppe2807/aseproject2425

Images:

giuseppe2807/aseproject2425:latest

Steps Taken:


1. **Building the images using Docker Compose:** *docker-compose build*
2. **Tagging the images for the repository, repeated for each service image:**
docker tag src-{service_image}:latest giuseppe2807/aseproject2425:latest
3. **Logging in to Docker Hub:** *docker login -u giuseppe2807*

After entering the username, the password was provided to authenticate.

4. **Pushing the images to the repository:** *docker push giuseppe2807/aseproject2425:latest*

By completing these steps, the tagged images were successfully uploaded to the Docker Hub repository, making them available for further analysis and deployment. **Docker Scout** was then used to inspect and improve the security posture of these images.

Among the results provided by **Docker Scout**, a critical issue was identified related to the version of **OpenSSL** used in the base image. To resolve this issue, we updated the base image by pulling the latest version of python:3.12-slim. This simple update successfully addressed the OpenSSL vulnerability and improved the security posture of our Docker images as we can see in the image below.



giuseppe2807/aseproject2425:latest

View recommended base image fixes

MANIFEST DIGEST sha256:2a2c1b82d0740efb20904b19f10b80bfa7d89a14e51d68a41d7f6e65c7bd2027

Analyzed by

| | | | | | |
|------------------------|-----------------------------|--|---------------|-------------------------------|---------------------------------------|
| OS/ARCH linux/amd64 | COMPRESSED SIZE 139.6 MB | LAST PUSHED 8 minutes ago by giuseppe2807 | TYPE Image | VULNERABILITIES 0 0 0 29 0 | MANIFEST DIGEST sha256:2a2c1b82... |
|------------------------|-----------------------------|--|---------------|-------------------------------|---------------------------------------|

Image hierarchy

FROM debian:9003c49cd9dbe316280204ebc2dd5ac03e0bcd0583d2bfaa45fc943868502a...

FROM python:3.12-slim, 3.12-slim-bookworm, 3.12.8-slim, 3.12.8-slim-bookworm

ALL giuseppe2807/aseproject2425:latest

Vulnerabilities (29)

Package or CVE name

Fixable

Show excepted

Reset filters

| Package | Vulnerabilities |
|-------------------|-----------------|
| No results found. | |

Layers (17)

0 # debian.sh --arch 'amd64' out/ 'bookworm' '@1733097600' 28.23 MB

1 ENV PATH=/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr... 0 B

2 ENV LANG=C.UTF-8 0 B

3 RUN /bin/sh -c set -eux; apt-get update; apt-get install -y --no-install-re... 3.32 MB

4 ENV GPG_KEY=7169605F62C751356D054A26A821E680E5FA6305 0 B

5 ENV PYTHON_VERSION=3.12.8 0 B

Additional Features

Feature: Admin user_stories

As we have indicated in the user stories section, we have implemented **all admin user stories**, i.e. in addition to login/logout and gacha management **also those concerning user and market management**. These are marked as: **Additional feature** in the index field.

Feature: Admin Functionality Tests in Postman

- **What is this feature?**
Automated tests in Postman to verify the correct functioning of admin-related APIs, integrated into the integration_test scenario.
- **Why is it useful?**
Ensures that the functionalities are correctly implemented and the system behaves as expected, improving reliability and reducing the risk of errors.
- **How is it implemented?**
Tests are executed in Postman, automating the validation of admin APIs within the integration_test scenario.

Feature: Login Attempt Limiter

- **What is this feature?**
A mechanism to limit the number of login attempts for user and admin accounts to prevent brute force attacks.
- **Why is it useful?**
Enhances security by preventing repeated login attempts from malicious users. It reduces the risk of brute-force attacks and ensures system stability.
- **How is it implemented?**
 - Login attempts are tracked in Redis.
 - The system counts attempts and applies a limit (e.g., 5 attempts).
 - If the limit is exceeded, a 400 error is returned until the Redis entry expires (e.g., 10 minutes as configured).