






ASE Project 24/25: Final Report

Group: SPIN&GO

Members:

- **Giuffrè Matteo**
- **Neglia Giovanni**
- **Staniscia Nicola**

Gachas Overview

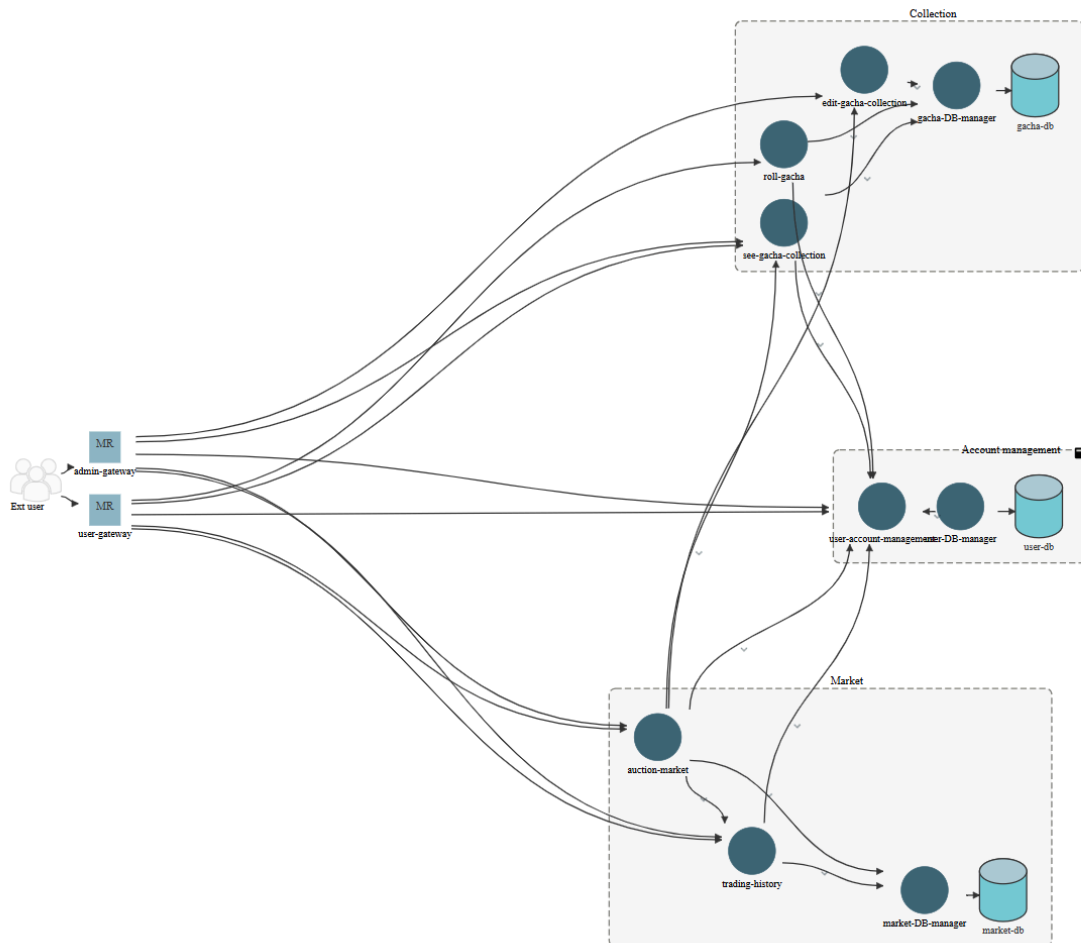
		
Common	Rare	Super Rare
		
Ultra Rare	Super Ultra Rare	

In our application, we used cartoon-style weapons with different rarity levels, including 18 common, 12 rare, 3 super rare, 2 ultra rare, and 1 super ultra rare.

The images of all Gachas are available in the **resources** directory:
/docs/resources

Architecture

The general Architecture of the Application is the following:



The system is composed of the following microservices:

- **Account_Management:**
 - This microservice handles the management of user and admin accounts. It allows the execution of main account-related operations such as: creating a new user, logging in and out, changing the password, etc.

- It has been implemented using Python and the libraries Flask, Flask-JWT-Extended, and Requests.
- **See_Gacha_Collection:**
 - This microservice handles the visualization of gachas and collections by both users and admins. It enables operations such as: viewing one's own collection (user), the system collection, all user collections (admin), and so on.
 - It has been implemented using Python and the libraries Flask, Flask-JWT-Extended, and Requests.
- **Edit_Gacha_Collection:**
 - This microservice handles the editing of user gacha collections and system gachas. It allows operations like editing user collections, editing system collections, and more.
 - It has been implemented using Python and the libraries Flask, Flask-JWT-Extended, and Requests.
- **Roll_Gacha:**
 - This microservice manages gacha rolls (draws). Users use it to acquire new gachas for their collection. There are three types of rolls: standard, gold, and platinum.
 - It has been implemented using Python and the libraries Flask, Flask-JWT-Extended, and Requests.
- **Auction_Market:**
 - This microservice provides the implementation of the auction market. It includes endpoints used to view auctions, create them, and place bids, along with all the accompanying logic. Additionally, it manages auction closures and refund logic through a dedicated scheduler.
 - It has been implemented using Python and the libraries Flask, Flask-JWT-Extended, Datetime, and Requests.

- **Trading_History:**

- This microservice is used to complete transactions on the market and print transaction histories upon request.
- It has been implemented using Python and the libraries Flask, Flask-JWT-Extended, and Requests.

All DBs are MySQL DBs and gateways are developed using Nginx.

Connection between microservices

The **Roll_Gacha** microservice is connected to **Account_Management** because it needs to verify that a user has sufficient currency to perform a roll. Moreover, it must deduct the currency after a successful roll execution.

The **See_Gacha_Collection** microservice is connected to **Account_Management** because the admin needs to associate users' usernames with their respective collections.

The **Auction_Market** microservice is connected to **Account_Management** because it needs to access a user's currency to verify if they can place a bid. Additionally, it must deduct the currency once the bid is successfully placed.

The **Trading_History** microservice is connected to **Account_Management** to process refunds when a bid is outbid and to transfer funds to the user who created the auction once it has ended.

The **Auction_Market** microservice is connected to **See_Gacha_Collection** to finalize the creation of an auction, as it needs to verify whether the user owns the specified gacha.

The **Auction_Market** microservice is connected to **Edit_Gacha_Collection** because when an auction is concluded, the gacha is transferred to the winner of the auction.

User Stories

Account Management (User)

- 4) **/account_management/create_user_account** (User_Gateway, Account_management, User_db_manager, User_db)
- 5) **/account_management/delete_user_account/{username}** (User_Gateway, Account_management, User_db_manager, User_db)
- 6) **/account_management/modify_user_account** (User_Gateway, Account_management, User_db_manager, User_db)
- 7a) **/account_management/login** (User_Gateway, Account_management, User_db_manager, User_db)
- 7b) **/account_management/logout** (User_Gateway, Account_management, User_db_manager, User_db)
- 14) **/account_management/buy_in_game_currency** (User_Gateway, Account_management, User_db_manager, User_db)

Account Management (Admin)

- 4a) **/account_management/admin/login** (Admin_Gateway, Account_management, User_db_manager, User_db)
- 4b) **/account_management/logout** (Admin_Gateway, Account_management, User_db_manager, User_db)
- 5) **/account_management/admin/view_users (additional)** (Admin_Gateway, Account_management, User_db_manager, User_db)
- 6) **/account_management/admin/view_users?username={username} (additional)** (Admin_Gateway, Account_management, User_db_manager, User_db)
- 7) **/account_management/admin/check_payments_history/{username} (additional)**

(Admin_Gateway, Account_management, User_db_manager, User_db)

Collection (User)

- **9a) /collection**
(User_Gateway, See_Gacha_Collection, Collection_db_manager, Collection_db)
- **9b) /collection/grouped (additional)**
(User_Gateway, See_Gacha_Collection, Collection_db_manager, Collection_db)
- **10) /collection/{gacha_id}**
(User_Gateway, See_Gacha_Collection, Collection_db_manager, Collection_db)
- **11) /system_collection**
(User_Gateway, See_Gacha_Collection, Collection_db_manager, Collection_db)
- **12) /system_collection/{gacha_id}**
(User_Gateway, See_Gacha_Collection, Collection_db_manager, Collection_db)
- **13) /roll/standard**
(User_Gateway, Roll_Gacha, Account_management, Collection_db_manager, Users_db_manager, Collection_db, Users_db)
- **25) /roll/standard | /roll/gold | /roll/platinum (additional)**
(User_Gateway, Roll_Gacha, Account_management, Collection_db_manager, Users_db_manager, Collection_db, Users_db)

Collection (Admin)

- 9a) **/admin/collections (additional)**
(Admin_Gateway, See_Gacha_Collection, Account_management, Collection_db_manager, Users_db_manager, Collection_db, Users_db)
- 9b) **/admin/collections/{user_id} (additional)**
(Admin_Gateway, See_Gacha_Collection, Account_management, Collection_db_manager, Users_db_manager, Collection_db, Users_db)
- 10) **/admin/edit/collection (POST to add, PATCH to modify, DELETE to remove) (additional)**
(Admin_Gateway, Edit_Gacha_Collection, Collection_db_manager, Collection_db)
- 11) **/system_collection/{gacha_id} (additional)**
(Admin_Gateway, See_Gacha_Collection, Collection_db_manager, Collection_db)
- 12) **/admin/edit/gacha (POST to add, PATCH to modify, DELETE to remove) (additional)**
(Admin_Gateway, Edit_Gacha_Collection, Collection_db_manager, Collection_db)

Market (User)

- **16) /auction_market/market (GET)**
(User_Gateway, Auction_Market, Market_db_manager, Market_db)
- **17) /auction_market/market (POST)**
(User_Gateway, Auction_Market, Market_db_manager, Market_db, See_Gacha_Collection, Collection_db_manager, Collection_db)
- **18) /auction_market/market/bid**
(User_Gateway, Auction_Market, Account_management, Users_db_manager, Users_db, Market_db_manager, Market_db)
- **19) /trading_history/market/transaction_history?user_id={}**
(User_Gateway, Trading_History, Market_db_manager, Market_db)
- **20) /auction_market/market/auction_win**
(Auction_Market, Market_db_manager, Market_db, Edit_Gacha_Collection, Collection_db_manager, Collection_db)
- **21) /auction_market/market/auction_complete**
(Auction_Market, Market_db_manager, Market_db, Trading_History, Account_management, Users_db_manager, Users_db)
- **22) /auction_market/market/auction_refund**
(Auction_Market, Market_db_manager, Market_db, Trading_History, Account_management, Users_db_manager, Users_db)

Market (Admin)

- 13) **/auction_market/admin/market (additional)**
(Admin_Gateway, Auction_Market, Market_db_manager, Market_db)
- 14)
/auction_market/admin/market/specific_auction?auctionID={} GET (additional)
(Admin_Gateway, Auction_Market, Market_db_manager, Market_db)
- 15)
/auction_market/admin/market/specific_auction?auctionID={} PATCH (additional)
(Admin_Gateway, Auction_Market, Market_db_manager, Market_db)
- 16) **/auction_market/admin/market/history (additional)**
(Admin_Gateway, Auction_Market, Market_db_manager, Market_db)

Market Rules

Auction Creation

The only verification deemed necessary during the creation of an auction is to ensure that users cannot create multiple auctions for the same Gacha they own.

Placing Bids

The following limitations have been imposed on bidding in auctions:

- Users cannot place bids on behalf of other users.
- Rational bids (e.g., 5.1) are not allowed.
- Negative bids, bids lower than the current price, or bids exceeding the user's available points at the time are prohibited.
- Users are not allowed to place bids on auctions they themselves created.
- Additional bids are permitted even if the user is already the current highest bidder.

To manage last-second bids effectively, a scheduled task is used to provide a slight tolerance window for bids placed at the auction's closing (see details in subsequent sections).

“Active” Refunds

To allow users to place multiple simultaneous bids without encountering insufficient points issues, a scheduler task (apscheduler) is utilized to process refunds for non-winning bids every 30 seconds.

Final Auction Closure

The closure of auctions is also managed by the scheduler task, which performs the following actions:

- Updates the auction status to “closed.”

- Assigns the Gacha to the winning bidder.
- Records the transaction of purchase/sale between the final "buyer" and the "seller."
- Processes any remaining refunds that were not handled by the task managing “active” auctions.

Testing

Isolation Tests

Isolation tests were conducted on the microservices by creating mock functions that simulate the responses of the DB Managers. Each isolation test included simulations of the positive behaviors of every endpoint, as well as at least one negative behavior for each endpoint. The microservices tested were as follows:

- **account_management**
- **edit_gacha_collection**
- **see_gacha_collection**
- **roll_gacha**
- **auction_market**
- **trading_history**

Integration Tests

Integration tests were performed for each microservice by making requests through the API Gateway (Admin Gateway and User Gateway).

Performance Tests

The application's performance tests were conducted using Locust through the API Gateway. The tested endpoints belong to the following microservices:

- **edit_gacha_collection**
- **see_gacha_collection**
- **roll_gacha**
- **auction_market**
- **trading_history**

Security - Data

An example of sanitized input is the **new_currency** parameter in the **modify_user** endpoint of the **Account_Management** microservice, where the admin can modify a user's currency by sending a **username** and **new_currency**. The parameters are validated by ensuring that both are present and checking the type as follows:

```
if not username or not new_currency:
    return jsonify({"error": "username and new_currency are required"}),
400
try:
    new_currency = int(new_currency) # Prova a convertirlo in intero
except ValueError:
    return jsonify({"error": "'new_currency' must be an integer"}), 400
```

Additionally, **MySQL Connector**'s placeholders are used to execute the query securely, for example:

```
cursor.execute("UPDATE users SET in_game_currency = %s WHERE username = %s", (new_currency, username))
```

This approach ensures that the input is sanitized, the data is validated, and SQL injection is prevented by using parameterized queries.

In the application, the only data that is encrypted are the user passwords. When a user creates an account through the **/create_user_account** endpoint of the **Account_Management** microservice, the password is not saved in plain text. Instead, the hash of the password is saved, calculated using the entered password and a randomly generated salt. The hashed

password and the corresponding salt are then stored in the **user_db**. When a user wants to log in to their account, the **/login** endpoint simply retrieves the salt and the hashed password associated with the entered username from the database, computes the hash of the entered password with the corresponding salt, and finally compares them, granting access and the authentication token only in the case of a match.

Security - Authorization and Authentication

Our scenario is distributed because each microservice is capable of validating the token independently. When a user logs in successfully, receives a token, generated in the **account_management** microservice, in the /login endpoint, which remains valid for one hour and contains user parameters such as ID, username, and role (user or admin):

```
access_token = create_access_token(
    identity=str(user_id),
    additional_claims={
        "username": username,
        "role": 'user'
    }
)
```

The **secret_key** used to sign the token is retrieved by the microservices as a secret from the Docker Compose file, ensuring it is not exposed in plain text in the code.

```
# Configurazione della chiave segreta per JWT
app.config["JWT_SECRET_KEY"] = get_secret('/run/secrets/jwt_password')
app.config["JWT_ACCESS_TOKEN_EXPIRES"] = timedelta(hours=1)
app.config['JWT_TOKEN_LOCATION'] = ['headers']
```


All endpoints of the microservices that require authentication check if the received token is valid for the user requesting the operation. If valid, the operation is executed. In the absence of a token, or with an invalid token, access to the operation will be blocked.

An instance of an endpoint requiring authentication:

```
@app.route('/account_management/modify_user_account', methods=['PATCH'])
@jwt_required() # Richiede un token JWT valido
def modify_user_account():
    current_user = get_jwt()["username"] # Ottieni l'utente dal token
    role = get_jwt()["role"] # Ottieni il ruolo dal token
    data = request.get_json()
    username = data.get('username')
    new_password = data.get('new_password')
    # Controlla che l'utente stia modificando il proprio account
    if username != current_user or role != 'user':
        return jsonify({"error": "Unauthorized"}), 403
```

As can be seen from the code above, the **modify_user_account** endpoint requires a valid JWT token, meaning it must not be expired. After that, it checks that the token belongs to the user making the request and verifies that the user's role is valid.

Security - Analysis

Static Analysis

The static analysis performed with Bandit on the **account_management** microservice did not detect any vulnerabilities in the code. The analysis of the microservices related to the Collection (**edit_gacha_collection**, **see_gacha_collection**, **roll_gacha**) produced the following report:

Metrics:

Total lines of code: 850

Total lines skipped (#nosec): 0

hardcoded_sql_expressions: Possible SQL injection vector through string-based query construction.

Test ID: B608

Severity: MEDIUM

Confidence: LOW

CWE: [CWE-89](#)

File: [./src/collection/collection_db_manager/app.py](#)

Line number: 219

More info: https://bandit.readthedocs.io/en/1.8.0/plugins/b608_hardcoded_sql_expressions.html

```
218             # Query
219             query = f'UPDATE Gacha SET {', '.join(column)} WHERE id = %s'
220             cursor = conn.cursor()
```

hardcoded_sql_expressions: Possible SQL injection vector through string-based query construction.

Test ID: B608

Severity: MEDIUM

Confidence: LOW

CWE: [CWE-89](#)

File: [./src/collection/collection_db_manager/app.py](#)

Line number: 270

More info: https://bandit.readthedocs.io/en/1.8.0/plugins/b608_hardcoded_sql_expressions.html

```
269             # Query
270             query = f'UPDATE Owned SET {', '.join(column)} WHERE id = %s'
271             cursor = conn.cursor()
```

blacklist: Standard pseudo-random generators are not suitable for security/cryptographic purposes.

Test ID: B311

Severity: LOW

Confidence: HIGH

CWE: [CWE-330](#)

File: [./src/collection/roll_gacha/app.py](#)

Line number: 126

More info: https://bandit.readthedocs.io/en/1.8.0/blacklists/blacklist_calls.html#b311-random

```
125             # Extract a card
126             card = rnd.choices(ids, weights=new_probs)[0]
127
```

WARNING: Due to space limitations on the page, the Bandit report for the Collection is incomplete. For more details, please refer to the Actions section on the GitHub repository.

Following Bandit's analysis of the Collection, a manual verification was conducted to assess the potential issue of SQL Injection. The verification concluded that the reported vulnerability is currently not present. This is because the flagged functions receive data and dynamically construct queries exclusively using prepared statements.

The static analysis for the Market microservices generated the following report:

Metrics:

Total lines of code: 1743

Total lines skipped (#nosec): 0

blacklist: Standard pseudo-random generators are not suitable for security/cryptographic purposes.

Test ID: B311

Severity: LOW

Confidence: HIGH

CWE: [CWE-330](#)

File: [./src/market/auction_market/app/app_test.py](#)

Line number: 165

More info: https://bandit.readthedocs.io/en/1.8.0/blacklists/blacklist_calls.html#b311-random

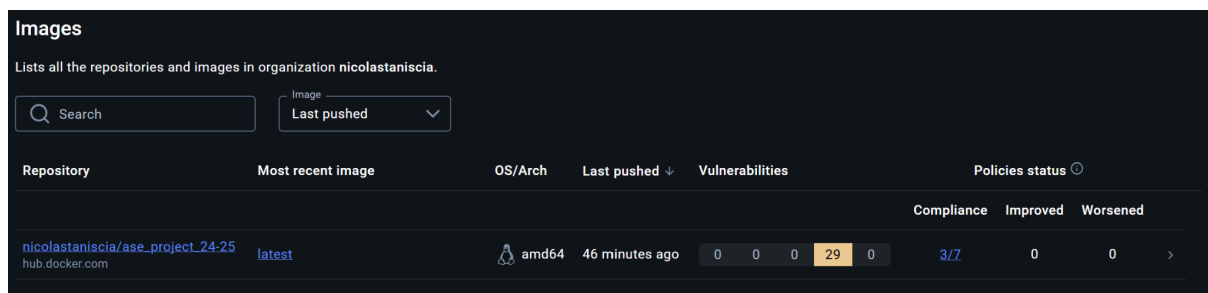
```
164         json_data={
165             "auction_id": (random.randint(100,500)),
166             "message": "Auction created successfully"
```

Additionally, the analysis of the Collection and Market identified an issue with the use of pseudo-random generators for cryptographic functions. However, it is important to clarify that the flagged functions are not used for cryptographic purposes but rather for random selections (such as drawing gacha items in **roll_gacha** and for testing purposes in **auction_market**).

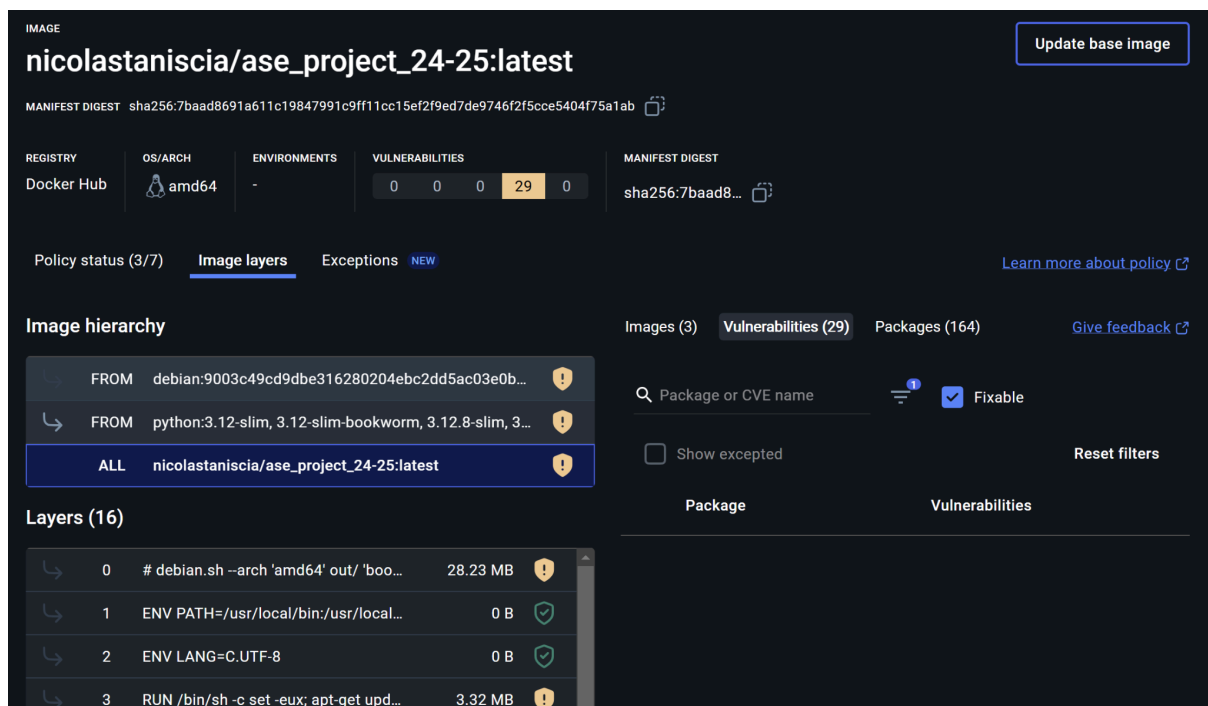
Dependency Analysis

The dependency analysis was conducted directly on the GitHub repository using Dependabot. The results of the analysis did not identify any vulnerabilities.

Docker Scout Analysis



The Docker Scout analysis of the images we built identified only 29 vulnerabilities, all of which were classified as low severity.



All 29 vulnerabilities, as shown in the image, are not fixable. For detailed information about these vulnerabilities, refer to the Actions section of the GitHub repository. There, a workflow is set up to push images to the Docker Repository and re-run the analysis with every push.

The name of the Docker Repository is:

nicolastaniscia/ase_project_24-25:latest

Additional features

DB init

To ensure true portability of MySQL, we included CSV files for database initialization. While working across different operating systems, we observed that MySQL, during initialization, configures certain options based on the host OS's file system. Specifically, the **lower_case_table_names** setting proved to be the key "problematic" configuration.

Grouped Collection

An additional feature has been implemented in the Collection, specifically in the **see_gacha_collection** microservice, through the endpoint **/collection/grouped**. This functionality was designed to address scenarios where a user might receive duplicate gacha items during a roll. Its purpose is to display the calling user's collection while adding an extra attribute, **quantity**, which represents the number of occurrences of each specific gacha item in the collection.