Advance Software Engineering Project
Final Submission

Group Members:

ASM Chafiullah Bhuiyan

Maheen Sabir

Javeria Kaleem

Samiuallah Khairy

Sayed Ala Moududi

Contents of the Report:

Game overview

There are several Gacha-Games available in the market, this one is inspired by one of the most popular online game League of Legends. The players are used as Gachas in this game. It was fun developing this project.

Gacha Overview

Each player is represented as a Gacha Card, with different rarity, price and inventory. The rarest gachas are expensive and inventory is low. Players can win them by rolling for a gacha or they can directly purchase from the system. Rolling is fun and gives you chances to win rare gachas at a very low price (not very cheap though 😉). Different gachas has different skills which makes the opponent weak if played tactically.

Rarity Levels:

Each gacha is categorized: 1-50/normal, 51-95/rare and 96-100/super rare gacha

- The roll price <= 50 we select a normal gacha based on random distribution
- The roll price <= 90 we select a rare gacha
- The roll price <= 100 then we select a super rare gacha

The game logic here is very simple, we didn't play that much with logic.

In-Game Currency: Runes

Sample Gachas:

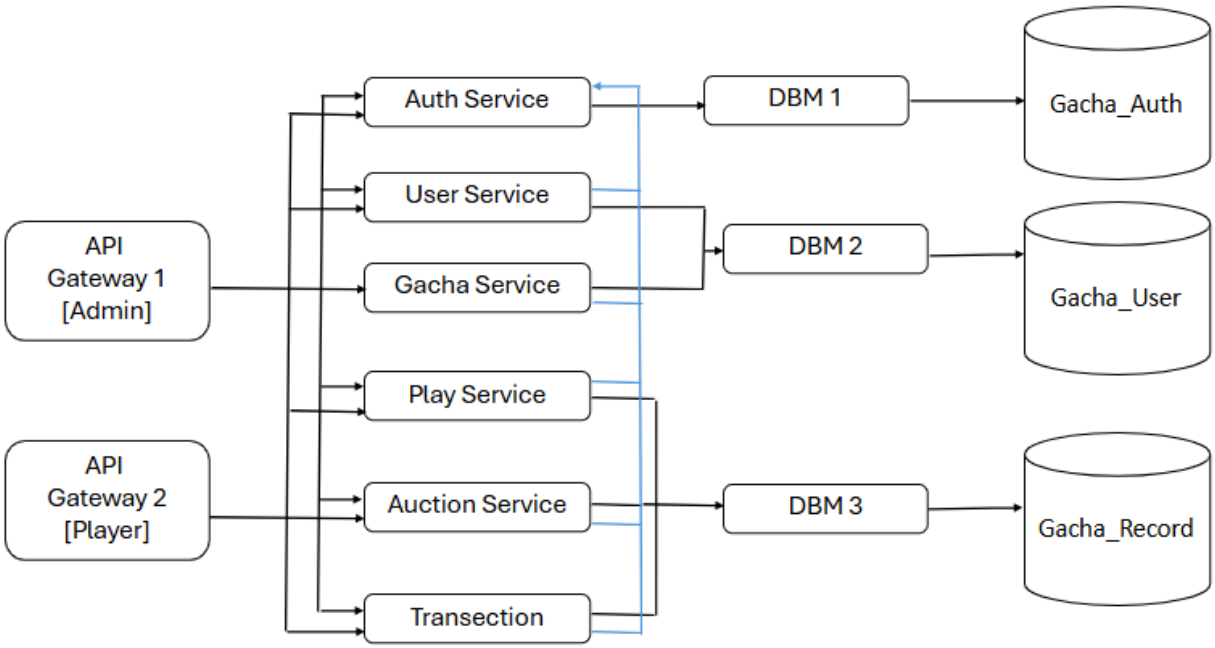
Name	Image	Rarity	Price
Annie		30	25
Akali		90	200
Atrox		25	20

Alister		**50**	**45**
Ashe		**60**	**50**
Anivia		**90**	**80**

Malzha		**40**	**30**
Lux		**99**	**100**
Yasuo		**85**	**100**

Xin Zhao		**70**	**75**

Architecture:



Description of the Micro-Services:

- **API Gateway 1(Admin Gateway)**

  The API Gateway 1 (Admin Gateway) encompasses both admin-specific endpoints and common endpoints shared between admin and player, including user creation, authentication, gacha management, player management, auction operations, and transaction processing but API Gateway one is only liable to the functions of an admin.

- **API Gateway 2 (Player Gateway)**

  The API Gateway 2 (Player Gateway) handles play services, auction operations, and transaction management for player-related activities.
  Acts as the entry point for all player-related requests, forwarding them to the appropriate microservices.

- **Auth Service**

  It is responsible for authentication and authorization, ensuring secure access by verifying user identities and granting appropriate permissions based on roles. Additionally, it manages user accounts, including creation, updates, deletion.

- **User Service**

  The User Service provides comprehensive admin and player account management, enabling the creation, modification, and deletion of accounts for both roles.
  Connected to **Auth-Service** for authentication and authorization purposes.

- **Gacha Service**

  The Gacha Service is responsible for creating, updating, and deleting gachas, as well as managing the gacha inventory within the system. It oversees the rarity and status of gachas, ensuring that legacy gachas are preserved in the system for historical reference and continuity. The service relates to **User service** so that we can preserve player gacha collections.

- **Play Service**

  The Play Service offers Roll to Win mechanism, where rarity depends on the roll price, and ensures one gacha per roll. It also governs Direct Purchases, verifying player status and balance, allowing each player to purchase one gacha only once. The service is connected to Gacha, User and Transaction service so that players can securely purchase their gacha keeping the transaction safe.

- **Auction Service**

The Auction Service enables admins to create and manage auctions, including setting start and end dates, and monitoring auction activity. It also enables players to place their gacha in running auctions. The service is connected to **User** and **Transaction Services to** ensure secure transactions between sellers and bidders, verifying auction completion, and preventing tampering with bids, while transferring gachas and adjusting player balances accordingly.

- ## Transection Service

The **Transaction Service** manages all financial transactions within the game, handling in-game currency debits and credits. It is integrated with the **User Service**, **Play Service**, and **Auction Service** to ensure accurate transaction processing, record transaction history for auditing purposes, and maintain consistency across all connected services.

- ## DBM1 (User Database Manager)

DBM1 is responsible for managing and maintaining databases related to user profiles and authentication data. Connected to the **Auth Service**, it securely stores user information such as usernames, passwords, roles, and authentication tokens. DBM1 ensures that user data is efficiently accessed and updated during login, registration, and profile management, while maintaining data security and integrity. It also supports role-based access control by storing user roles and permissions, enabling proper authentication and authorization for both admin and player accounts.

- ## DBM2 (Gacha Database Manager)

The DBM2 is responsible for managing and maintaining the database related to gacha items and rolls. It relates to both the **User Service** and **Gacha Service** to store critical metadata for each gacha item, including details such as rarity, availability, and other relevant attributes. DBM2 tracks gacha rolls, ensuring that items are correctly allocated to the right players based on the results. It also monitors and updates the inventory of available gachas, ensuring consistency across services and facilitating proper item distribution and availability.

- ## DBM3 (Transaction Database Manager)

The DBM3 is responsible for managing and maintaining the database related to transactions. It is connected to the **Auction Service, Play Service, and Transaction Service** to ensure seamless data storage, retrieval, and consistency across all services. The DBM3 ensures that all transaction-related data, including auction bids, in-game purchases, and financial records, are securely stored and easily accessible for processing, auditing, and reporting. It also ensures data integrity and consistency by coordinating with other services to reflect real-time updates in the database.

User Stories:

Role	Index	Endpoints	Microservices
player	4	api/player/user/create/	Gateway2, Auth-service, Database-one
player	5	api/player/user/ < int: user-id> /delete/	Gateway2, Auth-service, Database-one
player	6	Api/player/user/< int: user-id>/details/	Gateway2, Auth-service, Database-one
player	7, 8	/api/player/user/login/	Gateway2, Auth-service, Database-one
player	7	Api/player/user/ <int: user-id>/logout	Gateway2, Auth-service, Database-one
player	9	/api/player/play-service/player/<int:player_id>/collection/	Gateway2, play-service, Database-three
player	10	/api/player/play-service/player/collection/<int:collection_id>/	Gateway2, play-service, Database-three
player	11	/api/player/gacha/list/	Gateway2, gacha-service, Database-two
player	12	/api/player/gacha/< int: gacha_id>/details/	Gateway2, gacha-service, Database-two
player	13	/api/player/play-service/roll-to-win/	Gateway2, play-service, Database-three
player	14, 15	/api/player/transaction-service/player/<int:player_id>/purchase/game-currency/	Gateway2, Transaction service, Database-three
player	16	/api/player/auction-service/auction/list/	Gateway2, auction-service, Database-three
player	17	/api/player/auction-service/gachas/place/	Gateway2, an auction-service. Play-service, Database-three
player	18	/api/player/auction-service/gachas/<int:auction_gacha_id>/player/<int:player_id>/bid/	Gateway2, auction-service. play-service, user-service, Database-three
player	19	/api/player/transaction-service/player/<int:player_id>/all/	Gateway2, Transaction service, Database-three
Admin	20,21, 22,23	/api/admin/auction-service/gachas/<int:auction_gacha_id>/bids/winner/	Gateway1, auction-service, user-service, Database-three
Admin	4	/api/admin/user/login/	Gateway1, Auth-service, Database-one
Admin	4	/api/admin/user/<int:user_id>/logout/	Gateway1, Auth-service, Database-one
Admin	9	[get]/api/admin/play-service/player/<int:player_id>/collection/	Gateway1, play-service, Database-three

Admin	10	[put]api/admin/play-service/player/<int:player_id>/collection/	Gateway1, play-service, Database-three
Admin	11	[put]/api/admin/gacha/<int:id>/details/	Gateway1, Gacha-service, Database-two
Admin	12	[get]/api/admin/gacha/<int:id>/details/	Gateway1, Gacha-service, Database-two

Market Rules

General Description

The market in our gacha game allows players to engage in various activities such as rolling for gachas, directly purchasing gachas, and participating in auctions. The system is designed to balance fairness and competition while rewarding players who actively manage their resources. Below are the detailed rules and decisions for each market interaction.

Roll Prices

Players can roll for gachas at different price points:

- **Available Prices**: [50, 75, 95]
- **Rarity Mechanism**: Higher prices increase the likelihood of receiving a rarer gacha.
- **Requirements**: The gacha being rolled must be **active** and the inventory must be >= 2 because the system loves to preserve the last piece.

Direct Purchase

Players can directly purchase gachas from the market. But there are some conditions to meet:
- player has available balance
- inventory >= 2
- gacha already not belongs to player's collection
- gacha is active

Placing a Gacha on Auction

Players can place their gachas up for auction.

- The player setting up the auction determines the starting price.
- Only active gachas can be placed on auction.

Bidding Rules

Players can bid on active auctions to win gachas. The following rules apply:

General Rules

- A player cannot bid on an auction if they are the **seller** of the gacha.
- The **current balance** of a player must be ≥ the bidding price.
- A player's bid must exceed the **last bid price** to be valid.

End of Auction

- The **winner** is the highest bidder at the close of the auction.
- If the highest bidder's balance is insufficient to pay the bidding price at the end of the auction:
    - The next highest bidder will be announced as the winner.

Special Cases

1. **What happens to the currency of a player when someone else bids higher?**
    a. No currency is deducted until the player wins the auction. When another player bids higher, the previous bidder's currency remains unchanged.
2. **What happens if I bid at the last second of the auction?**
    a. The system will accept valid bids until the exact closing time. Last-second bids are accepted if they meet the rules.
3. **Can I bid on an auction in which I am the highest bidder?**
    a. No, a player cannot bid on an auction where they are already the highest bidder.

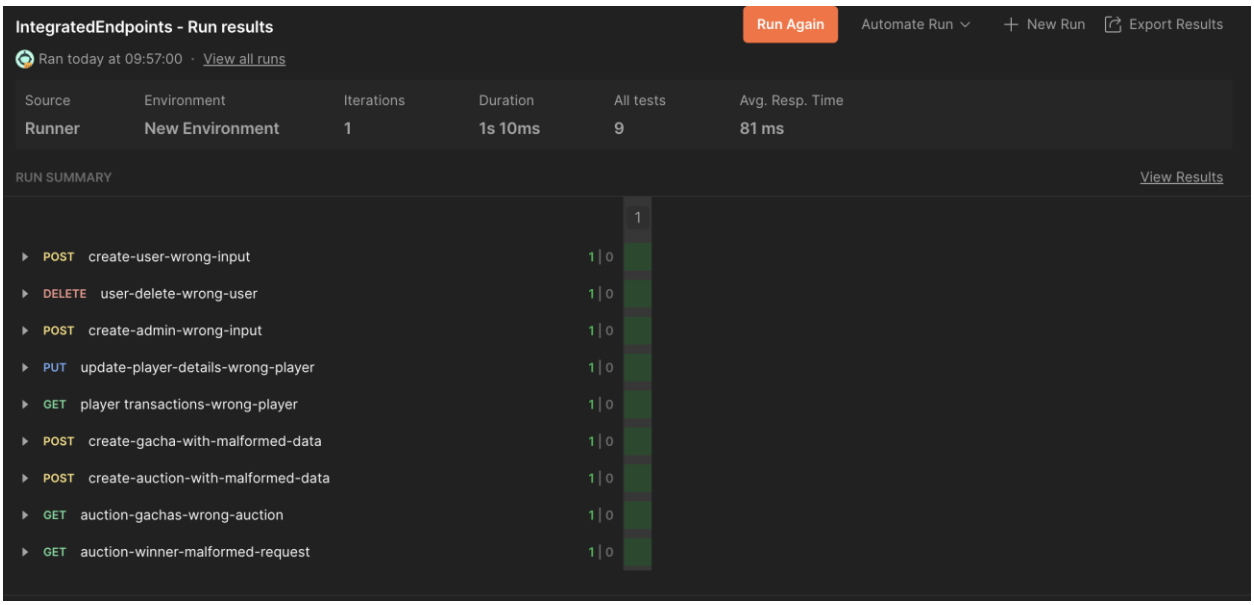
Currency Management

- Currency is only deducted when:
    - A player successfully rolls or purchases a gacha.
    - A player wins an auction.
- Players can replenish their balance through in-game currency purchases or rewards.

Testing

Integration Testing

We performed 3 types of testing in our project: Integration Testing, Isolation Testing and Performance Testing. Integration testing is performed via Postman using postman collection of each of the ApiGateways. We wrote the scripts to match our desired status code and message to ensure the operation is a success. We also performed some negative testing with malformed data to ensure that our sanitization works.

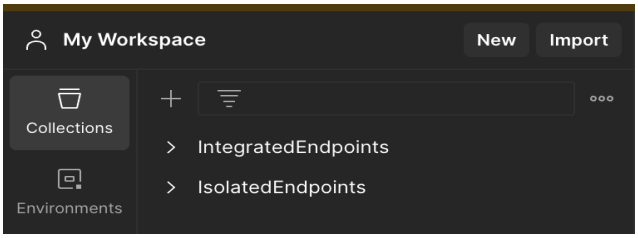


Isolation Testing

Services are also tested in isolation to ensure that they are working as expected with other services. There are two ways to test the services in isolation.

Via Postman:

Each service will be running on a separated port (just for testing) if you build your image with docker-compose.isolated.yml file. Then you have to import Services.json file from ApiDocs.



But remember one thing: if you want to test any service in isolation via postman you have to perform it as you are playing the game because this test will play with real data in the actual database. Well, that is not our proper way, it was just to make sure that the services are working in isolation as expected.

The actual isolation testing was done at application level for each of the services. For example we tested UserService with DBM_ONE to ensure we can successfully create the user and manage their information. We also tested PlayService with UserService, GachaService and TransactionService to ensure the market rules and secure transaction of player in game currency and gacha.

But in isolation the data has been mocked because generating tokens and interpreted data is hard while testing. For example to test roll to win mechanism we tested the /verify token and mocked the response as 200 which indicates the user has been authentication. For updating user balance, we mocked the response and balance to perform the transaction securely.

Here are some examples of data mocking:

```
mock_verify_token.return_value = True

# Mock user-service validation response
mock_get.return_value = Mock(
    status_code=200,
    json=Mock(return_value={
        "id": 1,
        "current_balance": 500
    })
)

# Mock external POST request
mock_post.return_value = Mock(
    status_code=200,
    json=Mock(return_value={"gacha_id": 1, "gacha_name": "RareGacha"})
)
```

Finally, you can run the tests individually:

```
docker exec -it 'container_name' /bin/sh
# For example if we want to test Gacha Service

docker exec -it gachaservice /bin/sh
python manage.py test
```

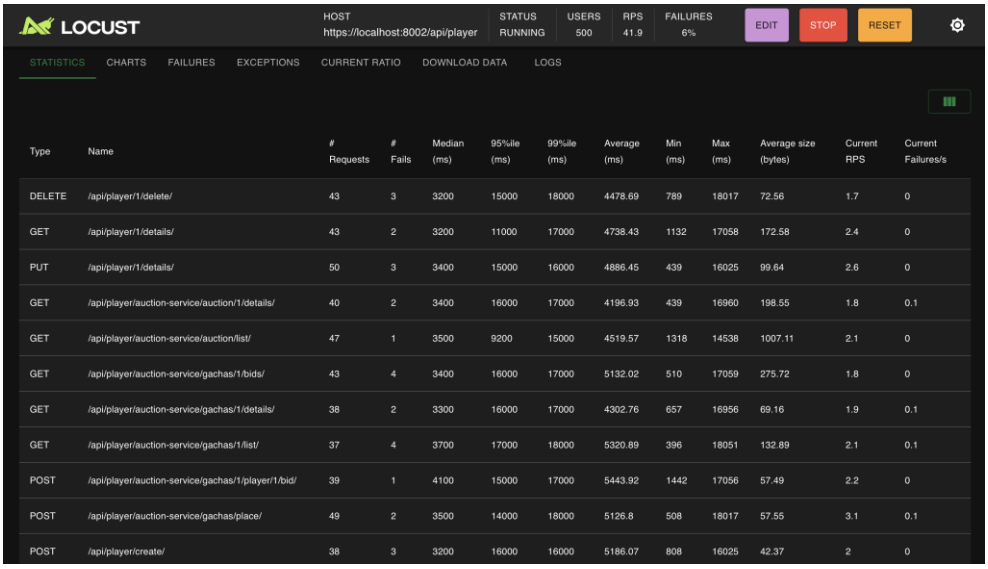
Performance Testing

To perform the testing, you must follow the steps mentioned in the git repo.

The final testing was the performance testing using locust where we tested all player endpoints in ApiGatewayTwo. For locust testing we have also mocked the auth tokens and expected responses. The procedure is explicitly mentioned in the readme.md in the repo.

```
common_headers = {
        "Authorization": "Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjozLCJ1c2VybmFtZSI6InBsYXllcjAwMSIsInJvbGUiOiJwbGF5ZXI
iLCJzdGF0dXMiOiJhY3RpdmUiLCJleHAiOjE3MzU5NDQ5OTMsImlhdCI6MTczMzM1Mjk5M30.YOvz2lUqAecFNhPnN6bYa6YmqD_q2pE
J1l1nb2Lx-C8",
        "Role": ','.join(['player']),
    }

def generate_random_string(self, length=8):
    """Generate a random string of fixed length."""
    return ''.join(random.choices(string.ascii_letters, k=length))
```



LOCUST

HOST https://localhost:8002/api/player | STATUS RUNNING | USERS 500 | RPS 41.9 | FAILURES 6% | EDIT | STOP | RESET

STATISTICS | CHARTS | FAILURES | EXCEPTIONS | CURRENT RATIO | DOWNLOAD DATA | LOGS

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
DELETE	/api/player/1/delete/	43	3	3200	15000	18000	4478.69	789	18017	72.56	1.7	0
GET	/api/player/1/details/	43	2	3200	11000	17000	4738.43	1132	17058	172.58	2.4	0
PUT	/api/player/1/details/	50	3	3400	15000	16000	4886.45	439	16025	99.64	2.6	0
GET	/api/player/auction-service/auction/1/details/	40	2	3400	16000	17000	4196.93	439	16960	198.55	1.8	0.1
GET	/api/player/auction-service/auction/list/	47	1	3500	9200	15000	4519.57	1318	14538	1007.11	2.1	0
GET	/api/player/auction-service/gachas/1/bids/	43	4	3400	16000	17000	5132.02	510	17059	275.72	1.8	0
GET	/api/player/auction-service/gachas/1/details/	38	2	3300	16000	17000	4302.76	657	16956	69.16	1.9	0.1
GET	/api/player/auction-service/gachas/1/list/	37	4	3700	17000	18000	5320.89	396	18051	132.89	2.1	0.1
POST	/api/player/auction-service/gachas/1/player/1/bid/	39	1	4100	15000	17000	5443.92	1442	17056	57.49	2.2	0
POST	/api/player/auction-service/gachas/place/	49	2	3500	14000	18000	5126.8	508	18017	57.55	3.1	0.1
POST	/api/player/create/	38	3	3200	16000	16000	5186.07	808	16025	42.37	2	0

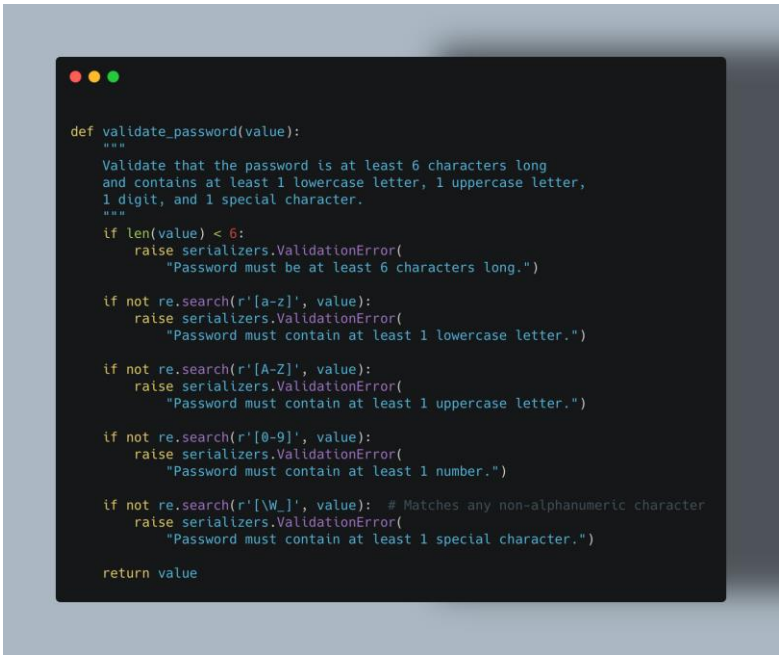
Data Security

To ensure robust data security, our implementation employs two critical measures: input sanitization and encryption. These steps collectively mitigate risks such as SQL injection and safeguard sensitive user information.

1. Input Sanitization

We sanitize all inputs, especially those more vulnerable to SQL injection and other malicious attacks, ensuring that only well-formed data reaches the server. This process applies strict validation rules to critical user information, such as usernames, passwords, and personal data. Key examples include:

- usernames are limited to a maximum of six characters to reduce the risk of buffer overflows or other injection-based attacks.
- phone numbers are restricted to exactly 10 numeric characters, ensuring consistency and preventing malformed inputs.
- bank details must conform to specific character constraints, explicitly disallowing special characters to prevent malicious data injection.
- passwords adhere to standard security guidelines, incorporating length and complexity requirements to ensure resilience against brute force and dictionary attacks.



```python
def validate_password(value):
    """
    Validate that the password is at least 6 characters long
    and contains at least 1 lowercase letter, 1 uppercase letter,
    1 digit, and 1 special character.
    """
    if len(value) < 6:
        raise serializers.ValidationError(
            "Password must be at least 6 characters long.")

    if not re.search(r'[a-z]', value):
        raise serializers.ValidationError(
            "Password must contain at least 1 lowercase letter.")

    if not re.search(r'[A-Z]', value):
        raise serializers.ValidationError(
            "Password must contain at least 1 uppercase letter.")

    if not re.search(r'[0-9]', value):
        raise serializers.ValidationError(
            "Password must contain at least 1 number.")

    if not re.search(r'[\W_]', value):  # Matches any non-alphanumeric character
        raise serializers.ValidationError(
            "Password must contain at least 1 special character.")

    return value
```

2. Input Encryption

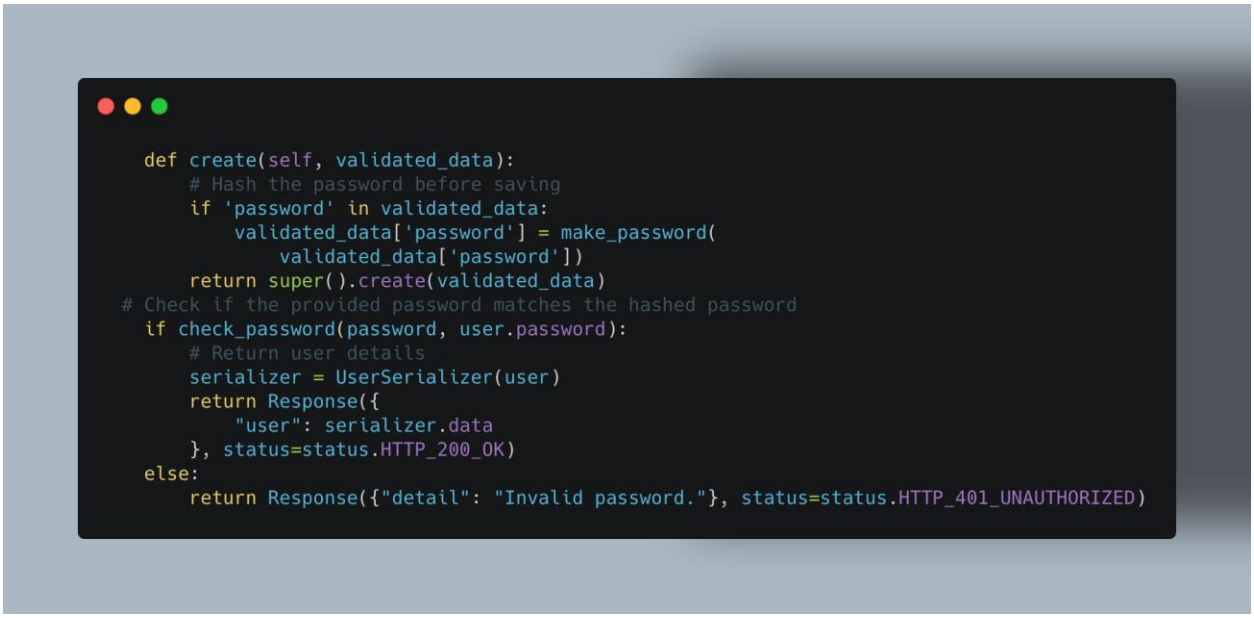
In our game we have encrypted 3 pieces of information regarding players so that the platform does not tamper with user information. Even in the case of data leaks we do not expose user information to the outside world. The information is following:

- Passwords
- Bank details
- Phone numbers

The encryptions are performed at application level, not at client level. The best practice is to encrypt data at client level so that the data enter encrypted to the network and the server is unaware of the data it is receiving. As we didn't implement any front-end our system is limited by that facility. But in our application, we maintained the standard procedure to encrypt data.

Encryption Process

We used two different approaches for passwords and personal information. For password encryption we use django's default password hashing technique (you can learn more here) which performs hashing by using PBKDF2 algorithm with a SHA256 hash, a password stretching mechanism recommended by NIST.



```
    def create(self, validated_data):
        # Hash the password before saving
        if 'password' in validated_data:
            validated_data['password'] = make_password(
                validated_data['password'])
        return super().create(validated_data)
# Check if the provided password matches the hashed password
if check_password(password, user.password):
    # Return user details
    serializer = UserSerializer(user)
    return Response({
        "user": serializer.data
    }, status=status.HTTP_200_OK)
else:
    return Response({"detail": "Invalid password."}, status=status.HTTP_401_UNAUTHORIZED)
```

We invoked two functions make_password() to hash the password and check_password() to check the hash with the given password. This process does not involve decrypting the whole password; thus, it is a secure one-way operation. Django hashes the given password and then performs check if both of the hash match together.

For user information we used another process as we need to decrypt our data to use it in several places. We used Fernet from Cryptographic library (you can read more here) to encrypt our user information. Encryption and decryption are performed using a helper function, which is called via the serializers and models to store and display data in proper format.

```python
from cryptography.fernet import Fernet
from django.conf import settings

# Generate a key once and store it securely (use Fernet.generate_key() to generate)
# Add your key to settings.SECRET_ENCRYPTION_KEY
encryption_key = settings.SECRET_ENCRYPTION_KEY
cipher = Fernet(encryption_key)


def encrypt_data(data):
    """Encrypt data using Fernet symmetric encryption."""
    if not data:
        return data
    return cipher.encrypt(data.encode()).decode()


def decrypt_data(encrypted_data):
    """Decrypt data using Fernet symmetric encryption."""
    if not encrypted_data:
        return encrypted_data
    return cipher.decrypt(encrypted_data.encode()).decode()
```

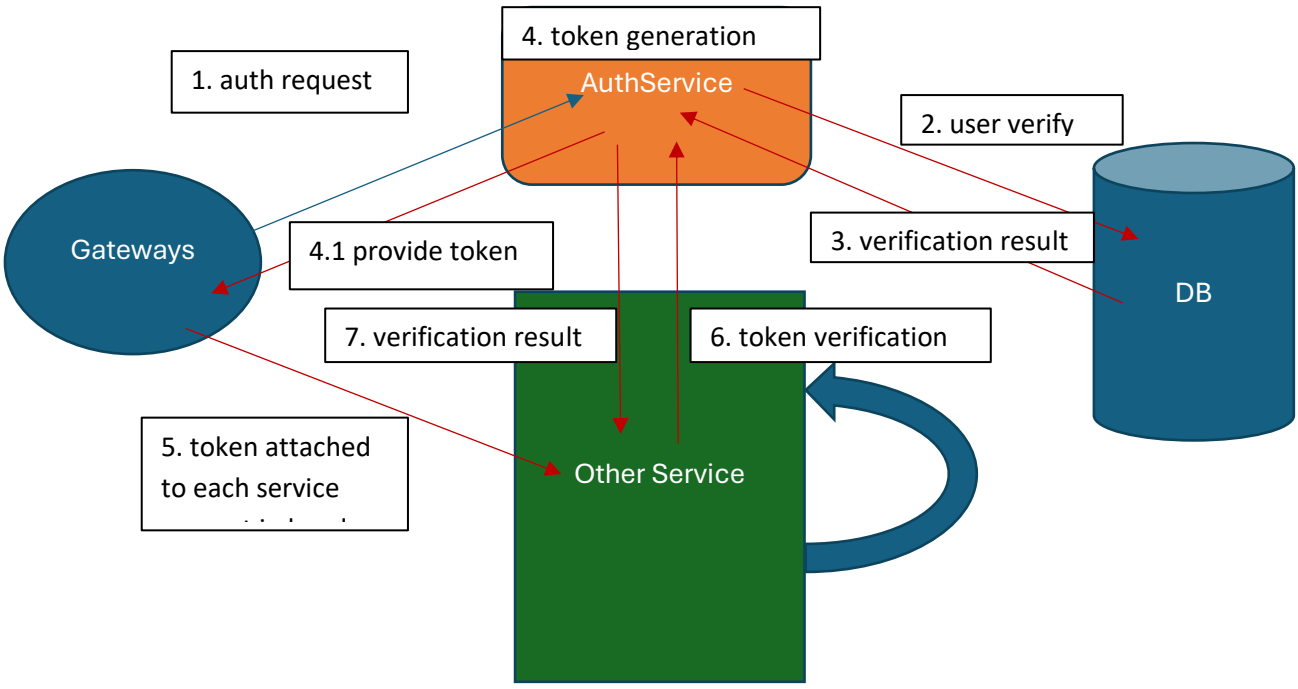
The salt is provided from the project settings file which is a 32 bit hash.

Authentication And Authorization

In our system we used a centralized authentication and authorization system. Auth Service is explicitly responsible for authenticating and authorizing our users. The authentication process follows the standard procedure of JWT tokens.

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. The representation consists in three Base64-URL-encoded strings separated by dots: Header, Payload and Signature (hash of encoded header and payload plus a private key) Encoding(Header.Payload.Signature).

We used PyJWT a well-adapted python library used by thousands of developers out there. PyJWT follows exactly the standards we mentioned above. Here is a simple overview of our authentication process:



Payload of the token:

json

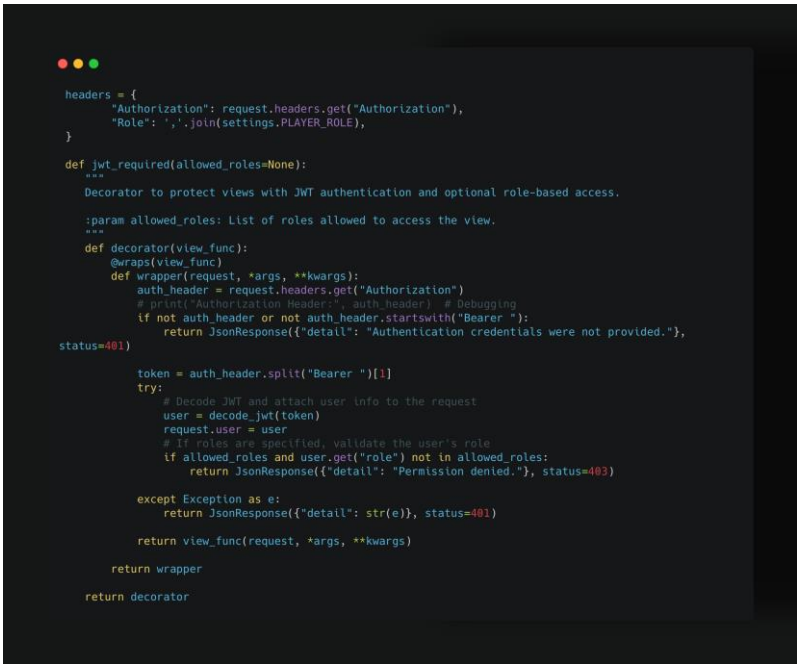
```
{
  "user_id": 1,
  "username": "admin001",
  "role": "admin",
  "status": "active",
  "exp": 1733428345,
  "iat": 1733410345
}
```

Payload Format Explanation

Key	Type	Description	Example Value
user_id	Integer	The unique ID of the user in your system.	1
username	String	The username associated with the authenticated user.	"admin001"
role	String	The role of the user, often used for access control.	"admin"
status	String	The status of the user account (e.g., active, inactive).	"active"
exp	Timestamp	The token's expiration time, expressed as a UNIX timestamp (seconds since 1970-01-01T00:00:00Z).	1733428345
iat	Timestamp	The time when the token was issued, expressed as a UNIX timestamp.	1733410345

The request is made by desired API_GATEWAYS and forwarded to AUTH_SERVICE. Initially the request is forwarded to DBM_ONE which is only response for storing authenticable information of a user. If the user records matches with the DB then the verification is forwarded to AuthService to generate auth tokens.

Later, we use the same token in our request headers to perform authorized access to our services.



```
headers = {
        "Authorization": request.headers.get("Authorization"),
        "Role": ','.join(settings.PLAYER_ROLE),
}

def jwt_required(allowed_roles=None):
    """
    Decorator to protect views with JWT authentication and optional role-based access.

    :param allowed_roles: List of roles allowed to access the view.
    """
    def decorator(view_func):
        @wraps(view_func)
        def wrapper(request, *args, **kwargs):
            auth_header = request.headers.get("Authorization")
            # print("Authorization Header:", auth_header)  # Debugging
            if not auth_header or not auth_header.startswith("Bearer "):
                return JsonResponse({"detail": "Authentication credentials were not provided."},
status=401)

            token = auth_header.split("Bearer ")[1]
            try:
                # Decode JWT and attach user info to the request
                user = decode_jwt(token)
                request.user = user
                # If roles are specified, validate the user's role
                if allowed_roles and user.get("role") not in allowed_roles:
                    return JsonResponse({"detail": "Permission denied."}, status=403)

            except Exception as e:
                return JsonResponse({"detail": str(e)}, status=401)

            return view_func(request, *args, **kwargs)

        return wrapper

    return decorator
```

Security Analysis

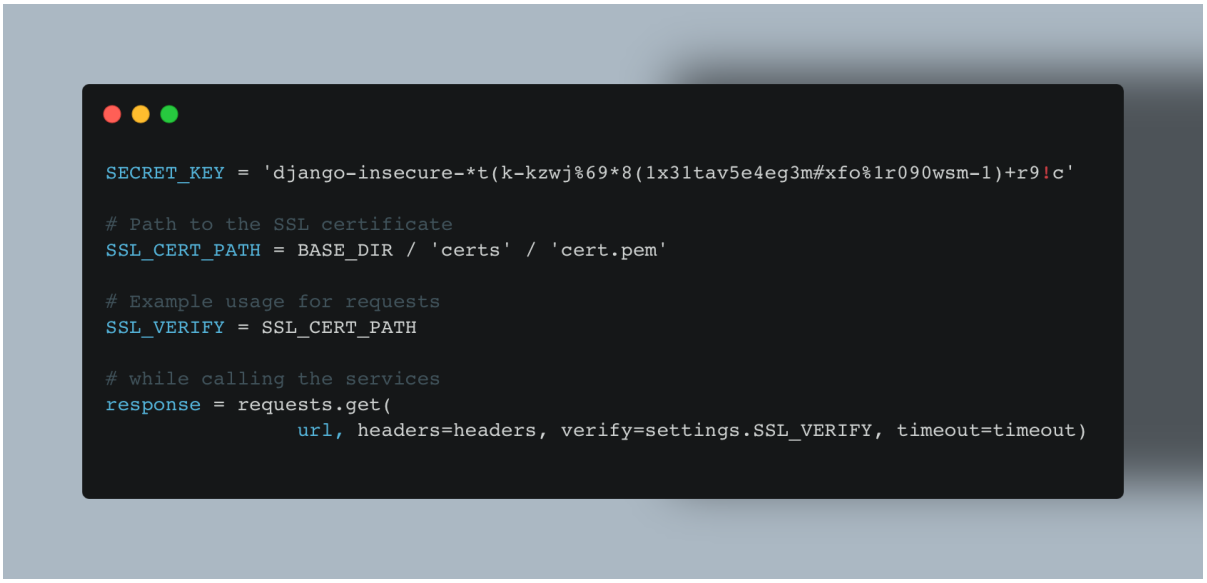
We performed two types of Security Analysis during the project. The first one is to speculate common vulnerability within our python codes. We used Bandit to perform this security analysis and for the first time we found several problems. For example,

1. Issue with SSL verification:

```
--------------------------------------------------
>> 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: .\ApiGatewayOne\api_gateway_one\views.py:30:23
29                  response = requests.post(
30                      url, json=data, headers=headers, verify=False, timeout=timeout)
31              elif method == "PUT":
32                  response = requests.put(

--------------------------------------------------
```

then we attached our self-signed certificate to the requests to avoid this vulnerability.



```
SECRET_KEY = 'django-insecure-*t(k-kzwj%69*8(1x31tav5e4eg3m#xfo%1r090wsm-1)+r9!c'

# Path to the SSL certificate
SSL_CERT_PATH = BASE_DIR / 'certs' / 'cert.pem'

# Example usage for requests
SSL_VERIFY = SSL_CERT_PATH

# while calling the services
response = requests.get(
                url, headers=headers, verify=settings.SSL_VERIFY, timeout=timeout)
```

2. Calling services without timeout:

```
--------------------------------------------------
>> Issue: [B113:request_without_timeout] Call to requests without timeout
   Severity: Medium   Confidence: Low
   CWE: CWE-400 (https://cwe.mitre.org/data/definitions/400.html)
   More Info: https://bandit.readthedocs.io/en/1.8.0/plugins/b113_request_without_timeout.html
   Location: .\ApiGatewayTwo\api_gateway_two\views.py:91:24
90              auth_service_url = f"{settings.AUTH_SERVICE}/{user_id}/details/"
91              auth_response = requests.get(auth_service_url, verify=False)
92
```

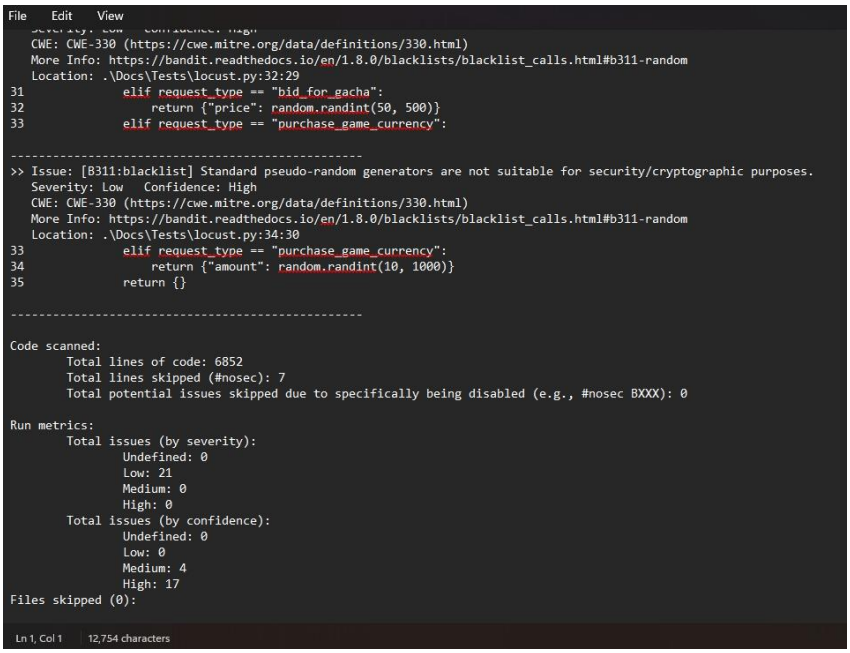
to solve this problem we added timeouts to our requests.

Then we found some problems with our secret keys(salts) as we hard-coded them in the settings.py, in product the practice is to generate a secret key while the application boots up using django-extensions package (python manage.py generate_secret_key). But for simplicity we ignored this issue for our project.

The last pieces of errors were due to locust.py file where we generated random strings to test the endpoints, so we ignored them as well.

```
--------------------------------------------------
>> 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: .\Docs\Tests\locust.py:32:29
31              elif request_type == "bid_for_gacha":
32                  return {"price": random.randint(50, 500)}
33              elif request_type == "purchase_game_currency":
```

Final report we got from Bandit:



Docker Scout Analysis

Most of the docker-scout problems were related to python version which is LOW so we ignored them. The problems reported as high was due to **setuptools** of python applications which we can solve by using <pip install --upgrade setuptools>. But if you are building your application with the latest version of python available, it's solved already.

Link to docker scout:
https://scout.docker.com/reports/org/samikhairy/images/host/hub.docker.com/repo/samikhairy%2Fgachagame