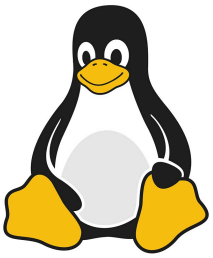
UNIVERSITÀ DI PISA

Report ASE Project 2024

SSRLostPity

Filippo Fornai - Luca Rizzo - Leonardo Scoppitto - Simone Stanganini

This page was intentionally left blank



Gachas

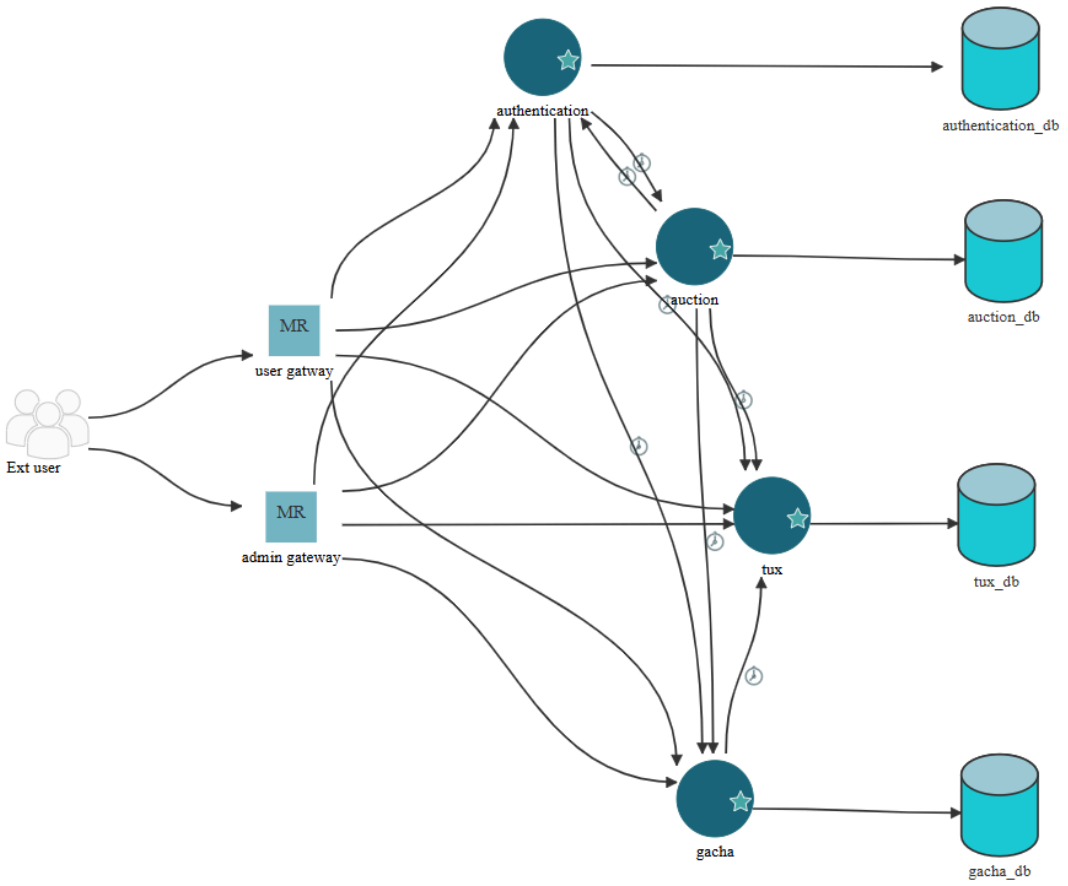
Our gacha system draws inspiration from a diverse range of Linux distributions and, the in-game currency used for the gacha system is named **TUX**, paying homage to the Linux mascot.

	Name	Rarity
	Alpine	★★★☆☆
	Arch	★★☆☆☆
	Bodhi	★★★★☆
	Debian	★☆☆☆☆
	Fedora	★☆☆☆☆
	Gentoo	★★★☆☆
	Hanna Montana Linux	★★★★★
	Kali	★☆☆☆☆
	Manjaro	★★☆☆☆
	Linux Mint	★☆☆☆☆
	NixOS	★★★★☆

	Name	Rarity
	openSUSE	★★☆☆☆
	Pop! OS	★★☆☆☆
	Raspberry Pi OS	★☆☆☆☆
	Slackware	★★★★☆
	Solus	★★★★☆
	Tails	★★★★☆
	TempleOS	★★★★★
	Ubuntu	★☆☆☆☆
	Void	★★★★☆
	Zorin	★★☆☆☆

Architecture

Our architecture presents solely the smell of **endpoint-based interaction** due to the direct communication between the various microservices.



We created two API gateways: one for admin operations and another for public user interactions. These gateways are deployed on separate Docker networks to ensure isolation. Ideally, the admin gateway should only be accessible via VPN, serving as a secure back-office, while the user gateway handles external user requests.
This setup aims to enhance security by separating admin and public access.

Here's a brief description of all the services:

- **Authentication Service:** handles the user registration, authentication, and authorization. It implements the OAuth 2.0 protocol. It uses python and the Fast API framework, MongoDB for the database.
- **Auction Service:** handles the auctions on the gacha marks. This service is responsible for keeping track of the bids, the auction timeout, etc. It uses python and the Fast API framework, MongoDB for the database.

- **Gacha Service:** handles user rolls, allows users to view the specifications of system gachas and their own gachas in their collection, and enables admins to modify, remove, or add gachas to the system. Additionally, it allows admins to manage each user's gachas individually. It uses python and the Fast API framework, MongoDB for the database.
- **Tux Service:** handles user to user payments at the end of the auctions, user to system when rolling a gacha, keeps track of the transactions, all the emitted TUX, etc. It also manages the freezing of the TUX when bidding in an auction. It uses python and the Fast API framework, Postgres for the database.
- **User and Admin API gateway:** they handle the call from outside the internal network from the users and the admin, they use the nginx reverse proxy.

List of connections and why

- **Gacha** Service to **Tux** service: the gacha service requests the payment of a roll for a specific user.
- **Authentication** service to **Auction** service: create the concept of user in auction market
- **Authentication** service to **Tux** service: creates a new balance with an initial amount of money.
- **Authentication** Service to **Gacha** Service: initializes an empty collection associated with the user to manage their future rolls and gachas.
- **Auction** Service to **Tux** service: checks whether a user has the correct amount of TUX to bid at an auction. It also settles all the payments when an auction closes. The admin can cancel an auction and the Tux service handles the reimbursement of the players.
- **Auction** Service to **Gacha** service: Auction Service notifies the Gacha Service to decrease the quantity of a gacha when a player creates an auction for it. Additionally, it notifies the Gacha Service to add the won gacha to the winner's collection when an auction concludes.
- **Auction** Service to **Authentication** service: calls to Gacha service and Tux service must be authenticated via a valid Admin token that is requested from Authentication service.

Simple mechanism of eventual consistency

In the **Authentication** Service, we adopted an **eventual consistency model** to manage the workflow for user creation and deletion, leveraging the saga pattern with the Authentication Service acting as the orchestrator (inspired by the "Saga Pattern" and "Splitting the Monolith" slides). This approach coordinates multiple transactions across local databases.
If a user creation or deletion fails in a downstream microservice, the operation of user creation/deletion is still reported as successful to the user. A background thread is initiated to retry communication with the affected microservice up to 5 times using an exponential backoff strategy. If all retries fail, a log entry highlights the need for manual compensation to restore consistency between services. While not currently implemented, automated compensation mechanisms could be introduced in the future to further enhance consistency.

User stories

1	AS A	player	I WANT TO	create my game account/profile	SO THAT	I can partecipate to the game
2	AS A	player	I WANT TO	delete my game account/profile	SO THAT	I can stop partecipating to the game
3	AS A	player	I WANT TO	modify my account/profile	SO THAT	so I can personalize my account/profile
4	AS A	player	I WANT TO	login and logout from the system	SO THAT	I can access and leave the game
5	AS A	player	I WANT TO	be safe about my account/profile data	SO THAT	nobody can enter in my account and steal/modify my info
6	AS A	player	I WANT TO	see my gacha collection	SO THAT	I know how many gacha I need to complete the collection
7	AS A	player	I WANT TO	want to see the info of a gacha of my collection	SO THAT	I can see all of info of one of my gacha
8	AS A	player	I WANT TO	see the system gacha collection	SO THAT	I know what I miss of my collection
9	AS A	player	I WANT TO	want to see the info of a system gacha	SO THAT	I can see the info of a gacha I miss
10	AS A	player	I WANT TO	use in-game currency to roll a gacha	SO THAT	I can increase my collection
11	AS A	player	I WANT TO	buy in-game currency	SO THAT	I can have more chances to win auctions
12	AS A	player	I WANT TO	be safe about the in-game currency transactions	SO THAT	my in-game currency is not wasted or stolen
13	AS A	player	I WANT TO	see the auction market	SO THAT	I can evaluate if buy/sell a gacha
14	AS A	player	I WANT TO	set an auction for one of my gacha	SO THAT	I can increase in game currency
15	AS A	player	I WANT TO	bid for a gacha from the market	SO THAT	I can increase my collection
16	AS A	player	I WANT TO	view my transaction history	SO THAT	I can track my market movement
17	AS A	player	I WANT TO	receive a gacha when I win an auction	SO THAT	only I have the gacha a bid for
18	AS A	player	I WANT TO	receive in-game currency when someone win my auction	SO THAT	the gacha sell works as I expect
19	AS A	player	I WANT TO	receive my in-game currency back when I lost an auction	SO THAT	my in-game currency is decreased only when I buy something
20	AS A	player	I WANT TO	that the auctions cannot be tampered	SO THAT	my in-game currency and collection are safe

1	POST	api/auth/accounts	Authentication, Gacha, Tux, Auction
2	DEL	api/auth/accounts/{{user_uid}}	Authentication, Gacha, Tux, Auction
3	PUT	api/auth/accounts/{{user_uid}}	Authentication
4	POST	api/auth/token	Authentication
5	******	**************************************	**************************************
6	GET	api/distro/{{user_uid}}/gacha/collection	Gacha
7	GET	api/distro/{{user_uid}}/gacha/collection /{{gacha_name}}	Gacha
8	GET	api/distro/user/gacha/all	Gacha
9	GET	api/distro/user/gacha/{{gacha_name}}	Gacha
10	POST	api/distro/{{user_uid}}/gacha/roll	Gacha, Tux
11	POST	api/tux-management/buy	Tux

12	******	**********	**********
13	GET	api/auctions	Auction
14	POST	api/auctions	Auction, Gacha, Authentication
15	POST	api/auctions/{auction_id}/bids	Auction, Tux, Authentication
16	GET	api/tux-management/transactions/{{user_uid}} (purchase, roll, won auctions)	Tux
17		**checkAuctionExpiration**	Auction, Tux, Gacha
18		**checkAuctionExpiration**	Auction, Tux, Gacha
19	POST	api/auctions/{auction_id}/bids (from another player)	Auction,Tux
20	******	**********	**********

CheckAuctionExpiration: a routine that is run at constant intervals that checks auctions that expired and handles both Tux and Gachas connected to them.

1	AS AN	administrator	I WANT TO	login and logout as admin from the system	SO THAT	I can access and leave the game
2	AS AN	administrator	I WANT TO	check all users accounts/profiles	SO THAT	I can monitor all the users accounts/profiles
3	AS AN	administrator	I WANT TO	check/modify a specific user account/profile	SO THAT	I can monitor and update a specific user account/profile
4	AS AN	administrator	I WANT TO	check a specific player currency transaction history	SO THAT	I can monitor the transactions of a player
5	AS AN	administrator	I WANT TO	check a specific player market history	SO THAT	I can monitor the market of a player
6	AS AN	administrator	I WANT TO	check all the gacha collection	SO THAT	I can check all the collection
7	AS AN	administrator	I WANT TO	modify the gacha collection	SO THAT	I can add/remove gachas
8	AS AN	administrator	I WANT TO	check a specific gacha	SO THAT	I can check the status of a gacha
9	AS AN	administrator	I WANT TO	modify a specific gacha information	SO THAT	I can modify the status of a gacha
10	AS AN	administrator	I WANT TO	see the auction market	SO THAT	I can monitor the auction market
11	AS AN	administrator	I WANT TO	see a specific auction	SO THAT	I can monitor a specific auction of the market
12	AS AN	administrator	I WANT TO	modify a specific auction	SO THAT	I can update the status of a specific auction
13	AS AN	administrator	I WANT TO	see the market history	SO THAT	I can check the market old auctions

1	POST	admin/api/auth/token	Authentication
2	GET	admin/api/auth/accounts	Authentication
3	POST GET	admin/api/auth/accounts/{{user_uid}}	Authentication

4	GET	admin/api/tux-management/transactions/{{user_uid}}	Tux
5	GET	admin/api/tux-management/transactions/{{user_uid}}	Tux
6	GET	admin/api/distro/gacha/all	Gacha
7	POST DEL	admin/api/distro/gacha admin/api/distro/gacha/remove/{gacha_name}	Gacha
8	GET	admin/api/distro/gacha/{gacha_name}	Gacha
9	PUT	admin/api/distro/gacha	Gacha
10	GET	admin/api/auction/auctions	Auction
11	GET GET	admin/api/auction/auctions/{auction_id} admin/api/auction/{auction_id}/bids	Auction
12	DEL PATCH	admin/api/auction/auctions/{auction_id} admin/api/auction/auctions/{auction_id}	Auction, Authentication, Tux, Gacha
13	GET	admin/api/auction/auctions?active=false	Auction

Market rules

- A player can use his Tux to place a bid on an active auction.
- To create an auction a player must own a copy of the specific gacha it wants to auction.
- Other players can participate in an auction only before the end of the auction that it's defined by the creator of the auction.
- A newly created auction has a starting price that is defined by the creator and that needs to be met to participate.
- Players can only bid higher than the current winning bid and when submitting a bid, the corresponding Tux amount is frozen by the tux-management service, preventing the user from spending them. The Tux of the user that was previously winning the auction is automatically released and deposited on its account.
- Whenever a bid is made it cannot be undone. This ensures fairness to other players that could possibly want to bid for the auction and avoids possible situations where two players cooperate to win the auction (ex. holding the winning bid high until the very last moment, then retracting it for another player to bid and win).
- Whenever an auction finishes the player that was winning wins the gacha and the player that created the auction receives the Tux that were offered.
- To ensure fairness players have visibility constraints on the auction market. Specifically they can only see the bids that they have done and the auctions that are currently active. This ensures players cannot auction and bid exploiting other players' patterns or analysis.
- No constraints are placed on the value of the gachas, however it is advised to consider the rarity of the gachas to place an appropriate starting price.

Testing

Unit tests - Test in isolation

To facilitate consistent testing and streamline GitHub workflow implementation, all microservices share a unified testing structure:

1. A `docker-compose` configuration runs a test version of the service image alongside a Newman container.
2. Newman executes Postman collections, validating the service's functionality.
    - The Newman container exits with code `0` if all tests pass, or code `1` otherwise.

This allows the implementation of a concise matrix job in the GH workflow in order to test the individual components in parallel, speeding up the testing process.

Authentication service

For isolated testing of the Authentication Service, we decided not to mock the database. Instead, we used a test-specific database instance included in the Docker Compose setup, which is launched as part of the test suite. However, we mocked the communication mechanisms with other services responsible for notifying the creation and deletion of users in the system.
To achieve this, we used the *unittest.mock* library, specifically the *patch* method. This allowed us to replace, within the scope of the main_test, the actual functions invoking other microservices with their mocked versions, ensuring the tests remain isolated from external dependencies. A dedicated Dockerfile was not created for testing purposes. Instead, a variable was set in the test's docker-compose configuration to specify which main file should be referenced. This variable is utilized by the existing Dockerfile, ensuring a single Dockerfile is used for both the application and the tests.

Auction service

To test Auction service in isolation a mocking strategy has been used. In particular:

- Calls to other services endpoints have been mocked to simulate their presence;
- Data that depends on randomization has been hardcoded during mocking to ensure a known state where endpoints can be tested (NB where possible it was avoided).
- Received authentication tokens are parsed through a substitute function that ignores the content of them and replaces the resulting parsed json with a dummy one.

To enable mocking Auction service is started via a support python script called “app_test.py”. This script sets the variable “mock_check” to “True” and substitutes the authentication function as said before.
The postman tests have been designed to run sequentially one after the other, letting the service navigate across known states given the hypothesis that we explained before.

Gacha service

For testing, we used a mocking strategy in the code to handle calls to other microservices, ensuring unit tests could run independently of external dependencies. Additionally, the random choice logic for selecting gacha was mocked, allowing predictable and repeatable tests. For integration tests, we leveraged Docker Compose with a `tmpfs` volume (`/data/db`) for the database, ensuring a fresh and clean state for every test run. This setup guarantees consistent results and prevents interference from leftover data.

Tux Service

The Tux service leverages a PostgreSQL database for production, but uses a lightweight SQLite database during tests to minimize overhead. Here's how it works:

- At startup, the service checks the `TEST_RUN` environment variable. If set, the service:
    1. Switches to an SQLite database created in a temporary directory.
    2. Creates all the tables to replicate the production database.
- The token verification logic is the only explicitly mocked function. Mocking is handled using the `@use_mocks` Python decorator, which checks the `TEST_RUN` variable. The decorator operates as follows:
    1. If a mock version of a function (e.g., `original_function_mock`) is defined in `libs.mocks`, it is used.
    2. If not, a `default_mock` function is executed.

```python
def use_mocks(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        if os.getenv("TEST_RUN", "false").lower() == "true":
                mock_func = getattr(
                        sys.modules["libs.mocks"],
                        f"{func.__name__}_mock",
                        default_mock
                 )
                return mock_func(*args, **kwargs)
        return func(*args, **kwargs)
    return wrapper

def default_mock(*args, **kwargs):
    pass

def example_mock(param):
   # do something mocked with param

@use_mocks
def example(param):
    pass
```

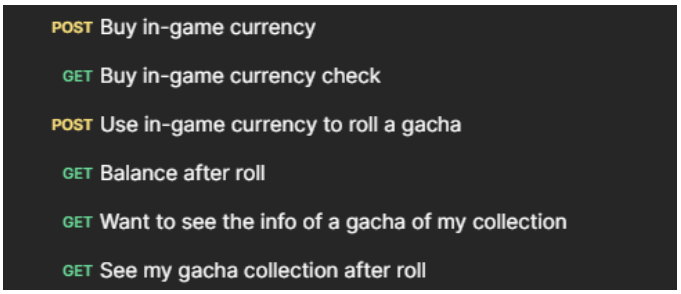
This approach ensures minimal overhead and a seamless switch between testing and production environments, providing full logic coverage without extensive function stubbing.

Integration test

In our integration tests, we focused on validating the interactions between microservices, ensuring seamless functionality across the system. **Each user story** was carefully mapped to its corresponding tests, providing comprehensive coverage of the application's features. Often, preliminary setup calls were required before testing the actual functionality of a feature. Additionally, further assertions were made to verify the overall consistency of the system through related API calls.

For instance, as shown in the following image, to perform a roll a setup step was needed to buy in-game currency. Only then the roll operation could be executed. Subsequent tests ensured that:

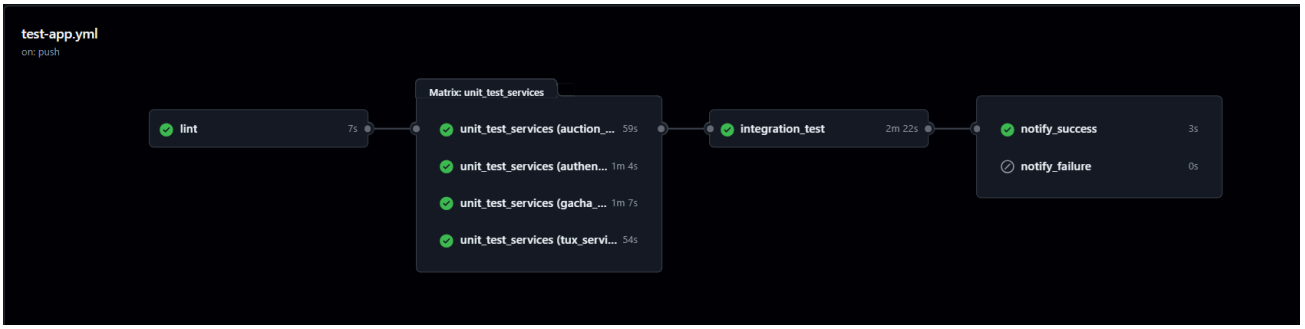
- The user's balance was correctly reduced after the roll.
- The rolled gacha was added to the user's collection.



CI/CD

Both the feature and the integration tests have been included in the github workflow so that at every push on the remote repository, the project gets tested and linted to spot both syntactic errors and bugs.

Below is an image of our GitHub test pipeline, highlighting the various jobs and the sequence in which they are executed. This pipeline runs on every push to all branches, ensuring that commits do not introduce any regressions. Specifically, we perform linting first, followed by isolated unit tests of each microservices, then integration tests, before finally notifying success or failure.



Locust - Performance tests

Below is the graph generated by Locust after a 15-minute run:

At the beginning of the test, response times are higher due to the user registration process, as we initialize a buffer of 6 users to simulate an active market. This ensures there are enough users to both roll gachas and participate in auctions right from the start. Users who receive a `402` error code during the `buy_tux` operation are removed from the user pool.

We can see that the average response time is quite low, while the following calls have a higher response time:

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	/api/auth/accounts	4140	8	2727.97	1175	8202	129.73	3.86	0.01
POST	/api/auction/auctions/7cc8d2b3-c852-4ef2-9136-b77bc7cdba6d/bids	9	0	2540.24	2126	3004	181	0.01	0
POST	/api/auction/auctions/2487b78a-1f50-4c0b-9158-31d97ae90659/bids	7	0	2525.87	2087	3138	181	0.01	0

As expected, the `/account` and `/auctions` endpoints are the most resource-intensive, because they have the most interactions with other services. In particular, the registration process involves the notification of the user data to all the other services, while the auction workflow includes the continuous interaction with the tux service and, at the end of the auction, with the gacha service. This overhead, especially in the case of the registration process, could be avoided by adopting a message broker that would have made the process asynchronous, but to avoid complexity we decided not to implement it.

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	/api/tux-management/buy	22202	23	34.41	8	876	41.19	20.68	0.02
POST	/api/auth/accounts	4140	8	2727.97	1175	8202	129.73	3.86	0.01
POST	/api/auction/auctions	346	0	2162.83	1368	2939	170.46	0.32	0
POST	/api/auction/auctions/00bccb9b-bf88-4f1b-a3f7-7b067702f963/bids	14	0	2091.93	1560	2821	181	0.01	0

As shown in the image above, our architecture demonstrates stability even under prolonged stress testing, with no significant failures observed. The only errors encountered involve the `/buy` endpoint, which occurs when users attempt to purchase Tux without sufficient funds, and the `/accounts` endpoint, triggered when users try to register with a username or email that already exists.

Security

Data

Authentication service

Just like the other services, the authentication server uses Pydantic models as an initial type-checking mechanism to validate the input data in requests. This ensures that basic data types, such as strings and integers, conform to the expected structure right at the entry point. Additionally, this service performs sanitization on various inputs retrieved from endpoints to prevent the injection of unwanted content into the database. Specifically, it enforces the following validations:

- **username**: Must contain only alphanumeric characters and, at most, the special character `_`.
- **email**: Must adhere to the standard email format (fully qualified and valid).

Additionally, a security check is performed on passwords to ensure robust protection. Passwords must meet the following criteria:

- Be at least 8 characters long.
- Include at least one uppercase letter.
- Include at least one lowercase letter.
- Include at least one number.
- Include at least one special character.

This ensures that data integrity and security are maintained while safeguarding against common vulnerabilities.

During registration, we also verify that the provided username and email do not already exist in the system to prevent conflicts and ensure unique user accounts.

For password hashing, we use **bcrypt**, which generates a single string containing all the necessary information: the algorithm version, cost factor, unique 22-character salt, and the hashed password. This approach ensures that we don’t need to store the salt separately, as it is embedded in the hash itself. With this self-contained string, we can validate passwords securely and efficiently.

Tux service

This service stores the user data in a postgres database. This choice was made to be sure that in case of errors or inconsistencies, the service is able to rollback the operations easily. In particular any function that stores, updates or deletes data from the database, its decorated with the transactional decorator:

```
def transactional(func):
      @wraps(func)
      def wrapper(session, *args, **kwargs):
      already_in_transaction = session.in_transaction()

      try:
            if not already_in_transaction:
                with session.begin():
                     return func(session, *args, **kwargs)
            else:
                  return func(session, *args, **kwargs)

      except Exception as e:
            logger.error(f"Error during transaction: {e}")
            if not already_in_transaction:
                  logger.info("Rolling back transaction...")
                  session.rollback()
            raise

      return wrapper
```

As we can see it checks whether a session is already in a transaction, otherwise it begins a transaction and in case of an error, automatically executes the rollback.

Finally, the library of choice to perform queries on the database it's SQLAlchemy as it provides an easy interface to avoid hardcoding any long SQL statement and provides automatic input sanitization when using the ORM (Object Relational Mapper) module to prevent injection attacks.

Regarding the auction handling, the checks performed are:

- When freezing the Tux of a user
    - User existence
    - The Tux amount value is positive and higher than the current highest bid
    - The user can afford to freeze that amount of Tux
- When settling an auction:
    - The winner is not the auctioneer
    - The users involved in the operation actually exist
    - The auction has not already been settled
    - The user is actually the highest bidder

For other CRUD operations, along with `NOT NULL` constraints where needed, the system performs the usual type and value checks for integers and floats.

Finally, just like the other services, Tux service uses pydantic models to validate requests' input data.

Gacha service

In the Gacha Service, two key inputs required sanitization: the gacha_name and its rarity. To ensure proper validation, Pydantic models were used. On top of these models we put character constraints on the input (e.g., trimming whitespace from gacha_name ecc..), and validate rarity against predefined values.

Auction service

- **Input sanitization:** the python module Pydantic has been used to sanitize the arguments of HTTP requests creating 5 BaseModels: Auction, AuctionOptional, Bid, BidOptional, AuthId. Specifically Auction and Bid define the body structure of the requests used to create the corresponding resources. It has to be pointed out that these two models do not contain all the fields that are saved inside the auctions database, just the ones that are necessary to create an auction. Instead AuctionOptional and BidOptional contain all attributes in "Optional" form. These last 2 models are used to perform search operations inside the database. Also url parameters and query parameters are statically typed so that when passed they are converted to the right type. Generally speaking every argument that is passed through the endpoints is converted to the right type. If the conversion fails it means it is not valid considering the constraints of the specific type (UUID,str,int).

- **Input validation:** endpoints require data that define entities across the entire system. In particular Auction service requires:
    - player_id: the id of a player
    - gacha_name: the name of a gacha

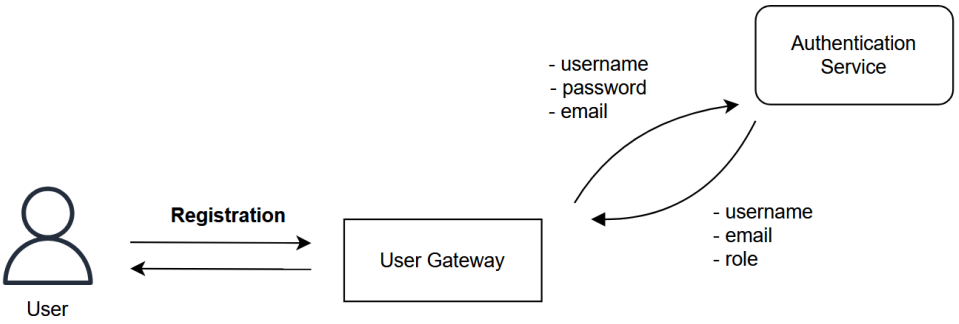
  To ensure the existence of the player the service exposes 2 endpoints: POST /users and DELETE /users/{{player_id}}. These endpoints are used by Authentication service to add and remove players from a internal database that Auction service keeps stored as a json. This ensures that whenever a player performs actions they are effectively part of the system.
  Instead gacha_name(s) are validated in a indirect way by calling Gacha service API to add and remove gachas. If the gacha_name does not exist the Gacha service will return an error, thus validating/invalidating the input.

Authorization and Authentication

We have chosen a distributed approach based on the OAuth 2.0 protocol to manage user authentication and authorization. The Authentication Service handles user registration and authentication by issuing **id_token** (to provide user identity information) and handles authorization by issuing **access_tokens** (to authorize API calls), following the **password grant flow** as defined in the OAuth 2.0 specification.

User Registration

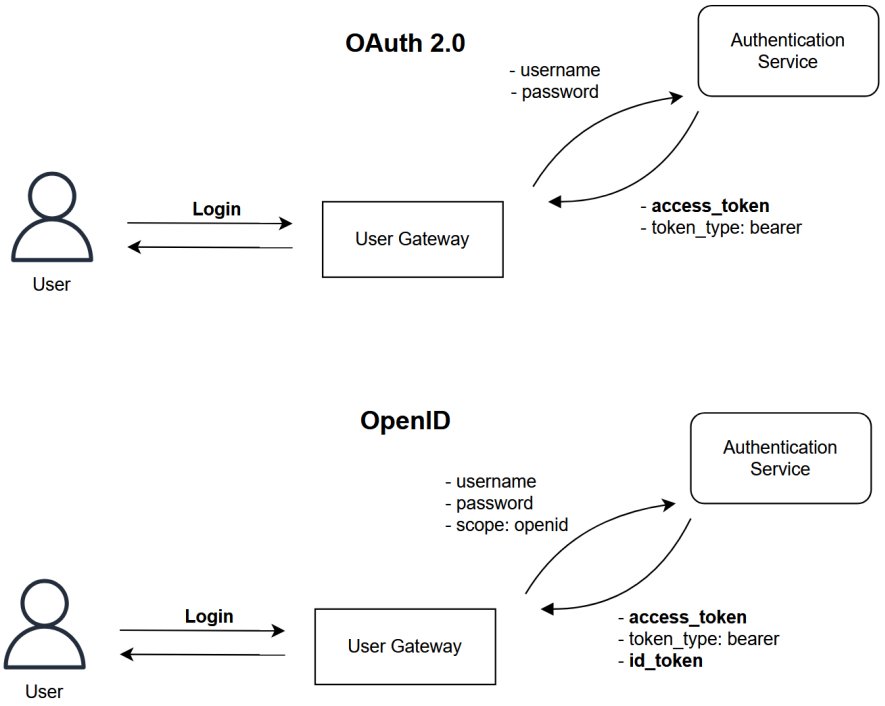
During registration, users provide their credentials and other necessary details (e.g., username, email, and password). The service validates this information (see data security section), stores it securely in the database, and hashes the password to ensure security against breaches.



Login Process

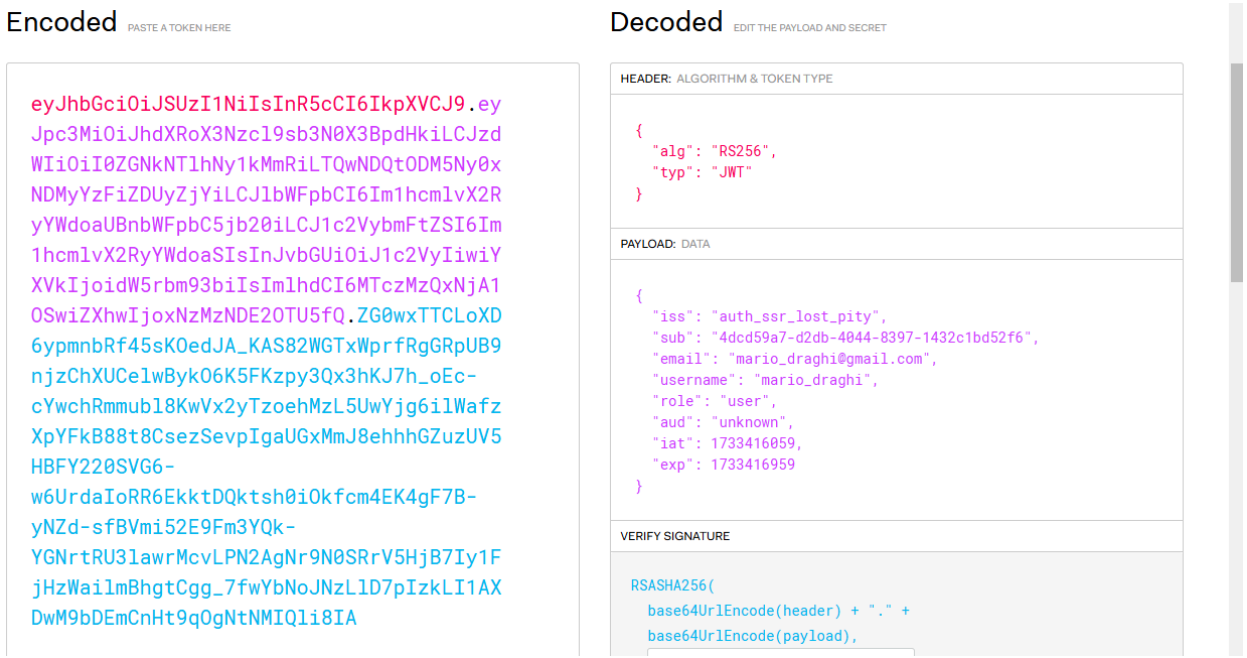
The login mechanism follows the **password grant flow** as required from project specification. In this flow, the client (in our testing, Postman) sends the username and password of the *resource owner* directly to the authentication service. The authentication service then validates the credentials and responds with the appropriate tokens based on client's needs:

- id_token: to obtain information about the user's identity (id_token)
- access_token: A token to authorize subsequent calls to other microservices (access_token)

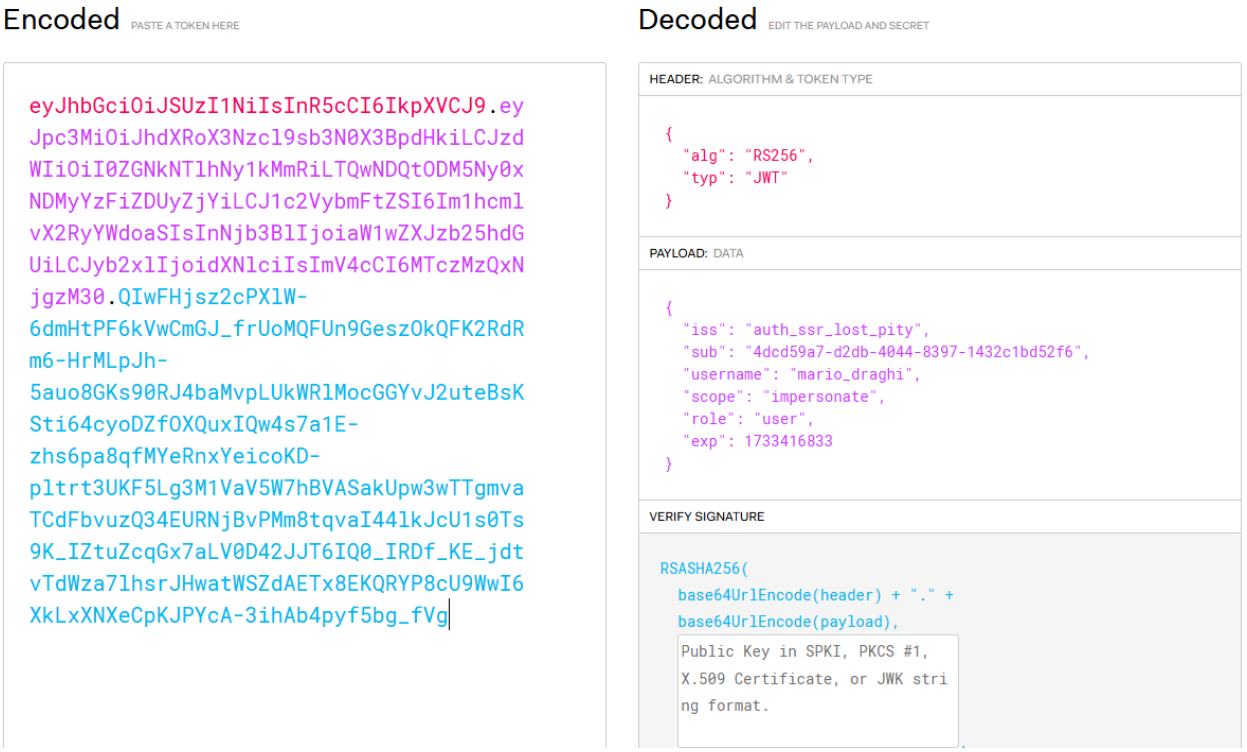


If the client specifies the ***openid*** scope in the login request, an id_token is returned alongside the access token. The id_token contains information about the user's identity, such as sub, email, and other attributes if stored on the server. The id_token also includes all the fields required by the OpenID Connect specification, such as iss (issuer), sub (user ID), aud (audience/client ID), exp (expiration time), and iat (issued at), ensuring compliance with the standard.

This token can be used client-side to identify the authenticated user. For example, if additional user information, such as first_name, last_name, or image_uri, were stored server-side, these could also be included in the JWT (JSON Web Token).
Here is an example of an ID token payload:



The **access_token** is used to make authorized calls to other microservices. It serves to indicate the client access rights. By default, the token includes a scope of "impersonate". This means the token grants permission to act as the user. Future developments could allow for more fine-grained access rights, enabling tokens tailored to specific roles or operations (e.g., limiting an admin user to only perform certain tasks).
Here is an example of an access token payload:



Token Signing Mechanism

The **access_token** and **id_token** are signed using an **asymmetric signing mechanism** (RS256). The authentication server signs the content with its private key, and any service or client can verify the token using the public key of the authentication server. For simplicity in our implementation, the public key is embedded as a secret, but in real-world scenarios, it is often exposed via a dedicated endpoint (e.g., /public-key). This approach ensures that tokens are tamper-proof while allowing distributed verification

Logout

Given the decentralized nature of the system, the logout functionality is currently **mocked**, relying on the short expiration time of the **access token**.

Future improvements could include broadcasting invalidated tokens to other services, enabling them to manage local blacklists and prevent any further use of the token.

Security Analysis

Bandit results

Execution of bandit on the entire codebase generated the following results:

Total lines of code: 2576
Total issues: 32

From the bandit analysis, we noticed that all the services shared these vulnerabilities:

- Issue: [B311:blacklist] Standard pseudo-random generators are not suitable for security/cryptographic purposes.
- Issue: [B501:request_with_no_cert_validation] Call to requests with verify=False disabling SSL certificate checks, security issue.
- Issue: [B104:hardcoded_bind_all_interfaces] Possible binding to all interfaces.

While only Tux service was affected by these one:

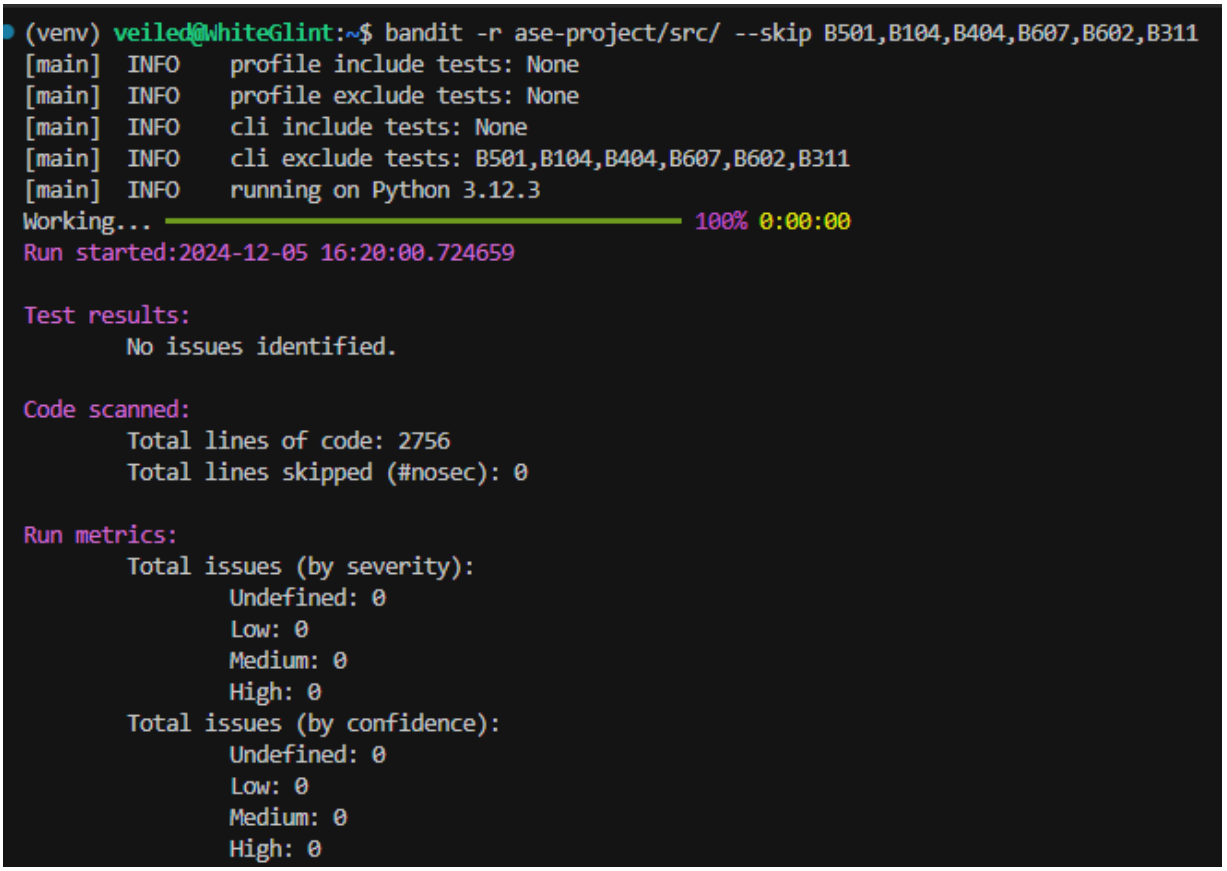
- Issue: [B607:start_process_with_partial_path] Starting a process with a partial executable path
- Issue: [B404:blacklist] Consider possible security implications associated with the subprocess module.
- Issue: [B602:subprocess_popen_with_shell_equals_true] subprocess call with shell=True identified, security issue.

(Risk level: low, medium, high)

B311, B501 and B104 are vulnerabilities that were considered during the project's development, as these services are intended solely for demonstration purposes.

B607, B404, and B602 are vulnerabilities relevant only to the Tux service, but are only confined to unit tests, where the `subprocess` module is imported to create the temporary directory.

With these risks accounted for, we re-ran Bandit, excluding these vulnerabilities, and achieved a result of zero issues (see screenshot on the next page).



Pip-audit results

Creating a virtual environment, installing every requirement of the 4 services and running pip-audit results in:

```
(venv) veiled@WhiteGlint:~$ pip-audit
No known vulnerabilities found
```

Following the list of modules that were scanned:

```
annotated-types:0.7.0|anyio:4.7.0|APScheduler:3.11.0|bandit:1.8.0|bcrypt:3.
2.2|boolean.py:4.0|CacheControl:0.14.1|certifi:2024.8.30|cffi:1.17.1|charse
t-normalizer:3.4.0|click:8.1.7|cryptography:44.0.0|cyclonedx-python-lib:7.6
.2|defusedxml:0.7.1|dnspython:2.7.0|email_validator:2.2.0|fastapi:0.115.6|f
astapi-cli:0.0.6|filelock:3.16.1|greenlet:3.1.1|h11:0.14.0|html5lib:1.1|htt
```

pcore:1.0.7|httptools:0.6.4|httpx:0.28.0|idna:3.10|Jinja2:3.1.4|license-expression:30.4.0|markdown-it-py:3.0.0|MarkupSafe:3.0.2|mdurl:0.1.2|msgpack:1.1.0|packageurl-python:0.16.0|packaging:24.2|passlib:1.7.4|pbr:6.1.0|pip:24.0|pip-api:0.0.34|pip_audit:2.7.3|pip-requirements-parser:32.0.1|psycopg2-binary:2.9.10|py-serializable:1.1.2|pycparser:2.22|pydantic:2.10.3|pydantic_core:2.27.1|Pygments:2.18.0|PyJWT:2.10.1|pymongo:4.10.1|pyparsing:3.2.0|python-dotenv:1.0.1|python-multipart:0.0.19|PyYAML:6.0.1|requests:2.32.3|rich:13.9.4|rich-toolkit:0.12.0|shellingham:1.5.4|six:1.17.0|sniffio:1.3.1|sortedcontainers:2.4.0|SQLAlchemy:2.0.36|starlette:0.41.3|stevedore:5.4.0|toml:0.10.2|typer:0.15.1|typing:3.7.4.3|typing_extensions:4.12.2|tzlocal:5.2|urllib3:2.2.3|uuid:1.30|uvicorn:0.32.1|uvloop:0.21.0|watchfiles:1.0.0|webencodings:0.5.1|websockets:14.1|

Docker scout results

The Docker Scout dashboard shows no vulnerabilities for the images associated with our microservices. Since the free plan allows enabling Docker Scout for only three repositories, the result for the image of the distro_service microservice was obtained using the command:

```
docker scout cves
```