ASE Project Documentation

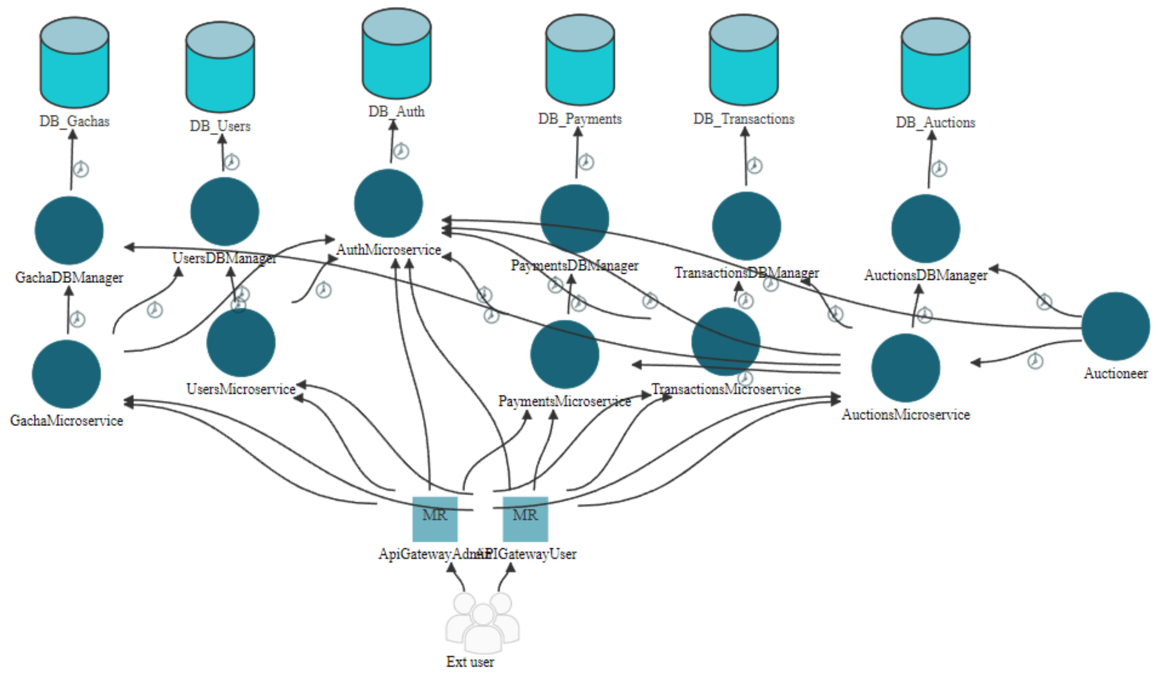
Group SRB

Advanced Software Engineering - Academic Year 2024-2025

Group Members
Tosic Natalija
Stevanovic Jana
Panic Veljko
Salmi Abderrahmane
Ceccotti Lorenzo

Architecture Overview

The backend system for managing the gachas is organized as a microservices architecture, where each microservice handles a specific feature of the system. This architecture is illustrated in the image below, which was generated using the microFreshner tool.



Microservices Overview

Microservice	Responsibility	Language & Technology
ApiGatewayAdmin	Handles all admin-related requests and routes them to the appropriate microservices.	Python (Flask)
ApiGatewayPlayer	Handles all player-related requests and routes them to the	Python (Flask)

	appropriate microservices.	
AuthMicroservice	Handles authentication operations like registration, login, and logout, generating and validating access tokens for both admins and players.	Python (Flask, PyJWT, SQLAlchemy)
AuctionDBManager	Manages the database for auctions, including creating, updating, and retrieving auction data.	Python (SQLAlchemy, Flask)
AuctionsMicroservice	Handles auction operations for both admins and players such as bidding and auction creation.	Python (Flask)
Auctioneer	Checks if there are any open auctions that need to be closed. If so it sends a request to the AuctionMicroservice to handle them.	Python
GachaDBManager	Manages the database for gacha items, including item creation, updates, and retrieval.	Python (SQLAlchemy, Flask)
GachaMicroservice	Provides player and admin functionalities for gacha operations, such as rolling and managing items.	Python (Flask)
PaymentsDBManager	Manages the database for in-game currency transactions, including purchases and refunds.	Python (SQLAlchemy, Flask)
PaymentsMicroservice	Handles in-game currency transactions, including player purchases and refunds.	Python (Flask)
TransactionsDBManager	Manages the database for player transaction history and records of in-game activities.	Python (SQLAlchemy, Flask)
TransactionsMicroservice	Handles player transaction records and ensures data	Python (Flask)

	consistency for all in-game activities.	
UsersDBManager	Manages the database for player and admin profiles, including authentication and profile updates.	Python (SQLAlchemy, Flask)
UsersMicroservice	Provides functionalities for player and admin account management, including registration and login.	Python (Flask)

Player Operations and Backend Flow

1. Get Player's Gacha Item Details

Description: This operation allows a player to view details of a specific Gacha item in their collection.

- **Flow**:
    1. **Step 1**: The player sends a `GET` request to the **API Gateway** at `/api/player/gacha/player-collection/{userId}/gacha/{gachaId}`.
    2. **Step 2**: The **API Gateway** forwards the request to the **Player Gacha Microservice**.
    3. **Step 3**: The **Player Gacha Microservice** sends a `GET` request to the **DB Manager** for Gacha collections at `/gachacollection/{userId}` to retrieve the player's Gacha collection.
    4. **Step 4**: The Gacha collection data is checked to see if `gachaId` exists in the player's collection.
    5. **Step 5**: If `gachaId` is found, the **Player Gacha Microservice** requests the item details by sending a `GET` request to the **DB Manager Gacha** service at `/gacha/{gachaId}`.
    6. **Step 6**: The **DB Manager Gacha** retrieves the Gacha item details and returns them to the **Player Gacha Microservice**.

    7. **Step 7**: The **Player Gacha Microservice** sends the response back to the **API Gateway**, which then returns it to the player.
- **Response**: The Gacha item details are returned to the player.

2. Roll a Gacha

Description: This operation allows a player to spend in-game currency to obtain a new random Gacha item.

- **Flow**:
    1. **Step 1**: The player sends a `POST` request to the **API Gateway** at `/api/player/gacha/roll` with their `userId`.
    2. **Step 2**: The **API Gateway** forwards the request to the **Player Gacha Microservice**.
    3. **Step 3**: The **Player Gacha Microservice** sends a `GET` request to the **DB Manager User** service at `/user/{userId}` to retrieve the player's details, specifically checking their in-game currency balance.
    4. **Step 4**: If the player's currency balance is sufficient, the **Player Gacha Microservice** proceeds with the roll by requesting a random Gacha item from the **DB Manager Gacha** service at `/gacha/random`; otherwise an error will be returned to the player.
    5. **Step 5**: After receiving the random Gacha item, the **Player Gacha Microservice** updates the player's in-game currency balance by sending a `PATCH` request to the **DB Manager User** service at `/user/{userId}`, deducting the roll price (which is a constant value saved in a json fileand mounted as a volume for this microservice).
    6. **Step 6**: The Gacha item is added to the player's collection by sending a `POST` request to the **DB Manager Gacha** service at `/gachacollection`, including `userId`, `gachaId`, and `source="ROLL"`.
    7. **Step 7**: The **Player Gacha Microservice** sends a success response back to the **API Gateway**, which then returns it to the player.
- **Response**: The rolled Gacha item is added to the player's collection, and confirmation is sent back to the player.

Operation: Purchase In-Game Currency

Description: This operation enables a player to purchase in-game currency by sending a specified amount of real money to the system. The backend verifies the request and updates the player's balance if the transaction is valid.

- **Flow**:
    1. **Step 1**: The player sends a `POST` request to the **API Gateway** at `/api/player/currency/purchase/` with their `userId` and `ingameAmount`.
    2. **Step 2**: The **API Gateway** forwards the request to the **Player Game Currency Microservice**.
    3. **Step 3**: The **Player Game Currency Microservice** validates the request payload, ensuring that both `userId` and `ingameAmount` are present and that `ingameAmount` is greater than zero.
    4. **Step 4**: The **Player Game Currency Microservice** sends a `GET` request to the **DB Manager User** service at `/user/{userId}` to confirm that the player exists in the system.
        - **If the player does not exist**, the **DB Manager User** service returns a 404 response, and the **Player Game Currency Microservice** returns an error response to the player through the **API Gateway**.
    5. **Step 5**: If the player exists, the **Player Game Currency Microservice** calculates the equivalent real amount to charge based on the `ingameAmount` and the currency rate defined in the system settings.
    6. **Step 6**: The **Player Game Currency Microservice** creates a transaction record by sending a `POST` request to the **DB Manager Payment** service at `/currencytransaction`, passing the details of the transaction including `userId`, `timeStamp`, `ingameAmount`, and `realAmount`.
        - **If the transaction is successful**, the **DB Manager Payment** returns a success message along with transaction details.
        - **If the transaction is invalid** (e.g., if there’s an error in processing), the **DB Manager Payment** responds with a 400 or 500 error code, which the **Player Game Currency Microservice** returns to the **API Gateway**.

7. **Step 7**: If successful, the **Player Game Currency Microservice** returns a confirmation of the purchase along with the transaction details to the **API Gateway**, which then relays it to the player.

- **Response**: The player receives a confirmation message along with the transaction details if the purchase is successful. If the purchase fails, an appropriate error message is returned based on the issue (e.g., user not found, invalid amount, or database errors).