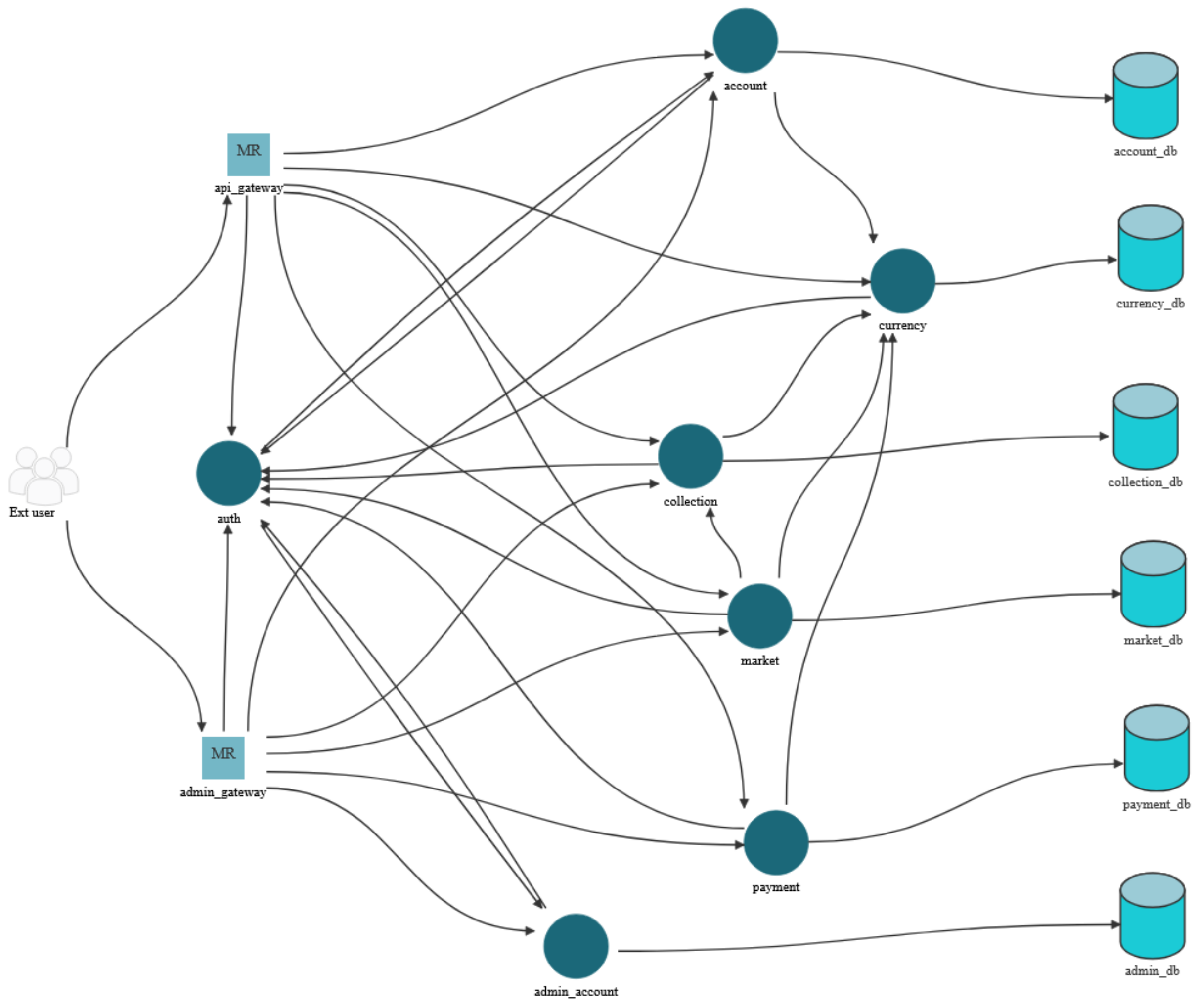
Architecture



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

1. **API Gateway**:
    - Acts as a single entry point for external users to interact with the microservices.
    - Routes requests to account, collection, market, currency, auth and payment for user API.
2. **Admin API Gateway**:
    - Acts as a single entry point for external admins to interact with the microservices.
    - Routes requests to account, admin_account, collection, market, auth and payment for admin API.
3. **Auth:**
    - Acts as authorization server in distributed authentication scenario.

- Every microservice (excepted for the gateway and admin gateway) when started send a request to auth to obtain the public key for token validation
- Interacts with account and admin_account to obtain user data for credential validation

4. **Account**:
   - Manages user accounts:
     1) users can create, delete e obtain their account
     2) admins can obtain all the user accounts, a specific by his user_id and change a specific user email
   - Interacts with account_db for storing user data and with currency for initialize user amount after user creation
5. **Admin Account**:
   - Manage admins accounts including creation and deletion.
   - Interacts with admin_db for storing admin data.
6. **Currency**:
   - Handles operations related to currency_amount related to the add, sub and get currency.
   - The only request exposed for API gateway allows users to check their currency_amount.
   - Interacts with currency_db for its data storage.
7. **Collection**:
   - Handles operations related to collection of gachas:
     1) users can obtain all type of system gachas, see the specific gacha by its item_id, see his collection and a specific item of his collection by its istance_id and roll gachas to add new items to his collection.
     2) admins can also see all type of system gachas and a specific gacha by its item_id, modify and delete an existing gacha and add a new gacha in the system.
   - Interacts with collection_db for storing collection-specific data and with currency in order to pay the roll operation.
8. **Market**:
   - Manages marketplace functionalities:
     1) users can get all open auctions in the market, buy a gacha bidding to a specific auction, sell one of his gacha creating an auction and see his transactions history
     2) admins can see all open auctions in the market, see the transactions history of all users or of one specific user, get and close a specific auction.

- Interacts with market_db for storing auction data, with currency for auction handling and with collection to check if a user is allowed to sell a specific gacha and to reassign it from the seller to the buyer.

9. **Payment Microservice**:
   - Handles payment transactions:
     1) users can buy new virtual coins in order to increase their currency_amount.
     2) admins can get all users' currency_history.
   - Interacts with payment_db for storing payment transactions.

10. **Databases (MySQL Instances)**:
    - Dedicated MySQL databases (*_db) for each microservice to ensure data isolation and modularity. These include:

      Collection Database (collection_db)
      Manages users' virtual collections. It includes two main tables:

      - **item**: Stores details about collectible items (ID, name, rarity, and image path).
      - **user_item**: Represents the relationship between users and items, tracking ownership and acquisition date.

      Account Database (account_db)
      Manages user account information. It contains one table:

      - **user**: Stores user account details (ID, username, email, and password) while ensuring the uniqueness of usernames and emails.

      Market Database (market_db)
      Supports the auction system for item sales. It includes one table:

      - **market**: Tracks auction records, including auction ID, seller, buyer (optional), start and end dates, status, and the current bid.

      Currency Database (currency_db)
      Manages users' virtual balance. It contains one table:

      - **currency**: Records users' virtual currency balance (ID and amount).

      Admin Database (admin_db)
      Handles system administrator accounts. It includes one table:

      - **admin_account**: Stores admin details (ID, username, email, and password) with unique constraints on usernames and emails.

      Payment Database (payment_db)
      Manages user payment transactions. It includes one table:

- **transactions**: Records transaction details (ID, user ID, card information, amounts, status, and timestamp).

The network configuration in the provided Docker Compose file divides the system into three distinct networks to isolate and manage service interactions effectively:

1. app-network:
    - A private network connecting all the microservices and their respective databases.
    - Ensures secure communication between internal services that are not exposed to the public.
2. public-network:
    - A network dedicated to services that need public access, such as the api_gateway.
    - Allows external clients to interact with the application through designated exposed ports.
3. admin-network:
    - A private network specifically for admin-related services, such as admin_api_gateway.
    - Isolates administrative functionality to enhance security, ensuring access is limited to authorized components.

Interesting player operations:

Viewing a Gacha in the Collection

If a player with id=1 wants to view a specific gacha in their collection with id=2:

1. The player sends a GET request to the API Gateway at /user/1/instance/2.
2. The API Gateway forwards the request to the collection microservice.
3. The collection microservice queries the collection_db database.
4. The collection_db filters the user_item table using the user's id and the item's instance id, retrieves the information, and returns it to the collection microservice.
5. The collection microservice returns the gacha's information (id=2) to the API Gateway.

Bidding in an Auction

If a player with id=1 wants to place a bid in an auction with id=3 for a gacha:

1. The player sends a PUT request to the API Gateway at /user/1/market/3/bid.

2. The API Gateway forwards the request to the market microservice.
3. The market microservice checks if the auction exists and if the bid is higher than the starting price or the current highest bid. It then sends a request to the currency microservice.
4. The currency microservice verifies if the player's balance is sufficient for the bid by sending a user/1/sub_amount request.
5. If the response is positive, the market microservice updates the bid for the auction with the player's bid and returns the previous bid to the player who had the highest bid (if applicable).
6. When the auction ends (based on the end_date attribute), the player obtains the gacha.

Putting a Gacha up for Auction

If a player with id=1 wants to put a gacha in their collection with id=2 up for auction:

1. The player sends a PUT request to the API Gateway at /user/1/instance/2/auction.
2. The API Gateway forwards the request to the market microservice.
3. The market microservice sends a request to /user/1/instance/2 to the collection microservice.
4. The collection microservice verifies that the gacha the player wants to put up for auction is in their collection and returns a status code.
5. If the response is positive, the market microservice creates a new auction with the player's gacha as the item.

Rolling a Gacha

If a player with id=1 wants to roll for a new gacha using their virtual coins:

1. The player sends a PUT request to the API Gateway at /user/1/roll.
2. The API Gateway forwards the request to the collection microservice.
3. The collection microservice sends a user/1/sub_amount request to the currency microservice.
4. The currency microservice checks if the player's balance is sufficient for the roll amount and returns a status code.
5. If the response is positive, the collection microservice generates a new gacha for the player.