[UDT Test] [Case Study] Design ERD for e-commerce

Status	Final
⌘ Areas	⌘ Side-projects
Archived	□
★ Favorite	□

Case study:

We are planning to build a backend for an e-commercial platform, your task is doing from system design, implementation to deploy production.

Question 2:

1. Design ERD for the backend of these features based on best practices to ensure scalable, easy coding. (3 points)

- Customer:
  - Can storage personal information (name, address, email, phone number, gender, etc…), cart, transaction, billing.
- Agency:
  - Can storage personal information (name, address, email, phone number, gender, etc…), product, transaction, billing.
- Admin:
  - Can read/create/update/delete agency.

Question 3:

1. With ERD you have already designed on question 1, what database are you using to implement? (1 point)
   Point)
2. Why are you using that? What is the strong and weak point of it? (1
3. Write docker-compose.yml to start the database locally. (1 Point) 4. Setup Spring (1 Point)
4. Using UML on question 1, set up API for these features. (5 Points)

Question 4:

1. Write a sequence diagram to build a solution for authentication and authorize adapt the list of features below. (5 Points)
   - Customer:
     Can log in, log out.
     Read transaction, billing linked with product.
     Read agency information.
   - Agency:
     Agency can read/create/update/delete of own product. Agency can read its own transactions, own billing.
   - Admin:
     Read billing, transaction, product, customer, agency. + Common:
     User can store auth state after reopening the browser.
2. Using the solution on question 1 implements these features. (15 Points)
3. What are the strong and weak points of your solution? How to improve that? (2 Points)
4. Build a solution for testing, ensure correct permission scalable from 100 APIS to 1000 APIs. (5 Points)

Question 5: Right now our application needs to synchronize products, pricing of the Agency by using third-party API.

1. Write a sequence diagram to build a solution to save, merge products data from third-party API to our database. (Third-party API data change every

hour) (8 Points)

2. What are the strong and weak points of your solution? How to improve

Answer:

Question 2: ERD for e-commerce

Tables and Their Purpose

1. **Accounts**
    - Manages all users in the system with common fields: `username`, `email`, `password`.
    - Includes an `account_type` field that categorizes each user as `ADMIN`, `CUSTOMER`, or `AGENCY`.
2. **Customers and Agencies**
    - **Customers**: Represents customer-specific details (e.g., `name`, `address`, `phone_number`, `gender`). Each customer is linked to an account via `account_id`.
    - **Agencies**: Stores agency-specific information with similar fields as customers and links to an account via `account_id`.
3. **Carts and Cart Items**
    - **Carts**: Represents shopping carts for customers. Linked to a customer through `customer_id`.
    - **Cart Items**: Contains items within a cart with fields for `quantity`, `status`, and references to `cart_id` and `product_model_id` to link the item to a specific product.
4. **Products and Product Models**
    - **Products**: Represents main product information provided by an agency. Linked to an agency through `agency_id`.
    - **Product Models**: Represents variations of a product (e.g., color, size, price). Each product model links to its parent product through `product_id`.

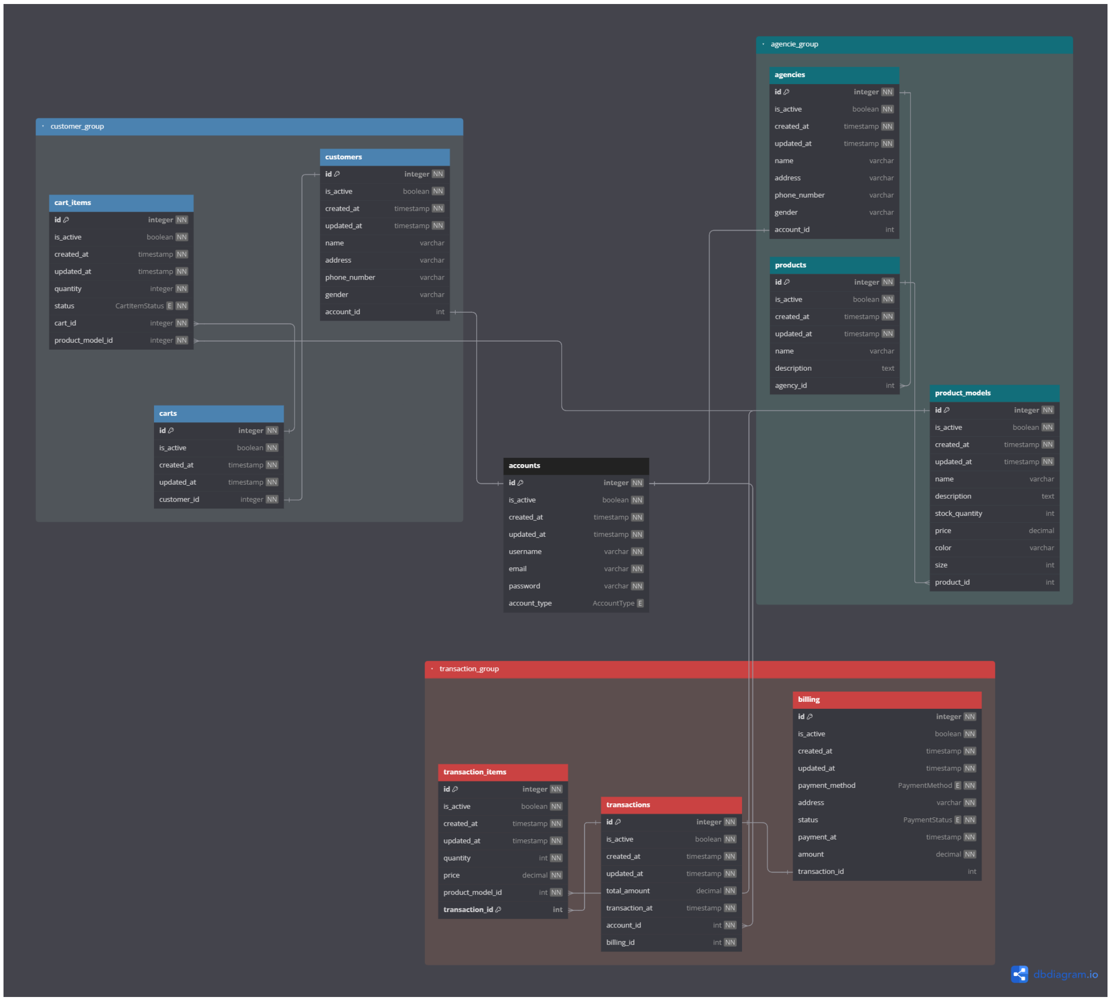
5. **Transactions and Transaction Items**
    - **Transactions**: Tracks purchases with fields like `total_amount` and `transaction_at` date, and is linked to both an `account_id` (buyer) and a `billing_id`.
    - **Transaction Items**: Details each item within a transaction, including `quantity`, `price`, and links to the specific `product_model_id` and `transaction_id`.
6. **Billing**
    - Manages billing and payment information for transactions. Includes fields like `payment_method`, `address`, `status`, and `amount`, linked to a transaction via `transaction_id`.

Enums

1. **AccountType**: Defines account types (`ADMIN`, `CUSTOMER`, `AGENCY`).
2. **CartItemStatus**: Status of items in the cart (`ACTIVE`, `PAYMENT_PENDING`, `PURCHASED`, `STOPPED`).
3. **PaymentMethod**: Payment options like `CREDIT` or `CASH`.
4. **PaymentStatus**: Payment status (`PENDING`, `SUCCESS`, `FAILED`).

ERD

Image:



customer_group

cart_items		
id	integer	NN
is_active	boolean	NN
created_at	timestamp	NN
updated_at	timestamp	NN
quantity	integer	NN
status	CartItemStatus E	NN
cart_id	integer	NN
product_model_id	integer	NN

customers		
id	integer	NN
is_active	boolean	NN
created_at	timestamp	NN
updated_at	timestamp	NN
name	varchar	
address	varchar	
phone_number	varchar	
gender	varchar	
account_id	int	

carts		
id	integer	NN
is_active	boolean	NN
created_at	timestamp	NN
updated_at	timestamp	NN
customer_id	integer	NN

agencie_group

agencies		
id	integer	NN
is_active	boolean	NN
created_at	timestamp	NN
updated_at	timestamp	NN
name	varchar	
address	varchar	
phone_number	varchar	
gender	varchar	
account_id	int	

products		
id	integer	NN
is_active	boolean	NN
created_at	timestamp	NN
updated_at	timestamp	NN
name	varchar	
description	text	
agency_id	int	

product_models		
id	integer	NN
is_active	boolean	NN
created_at	timestamp	NN
updated_at	timestamp	NN
name	varchar	
description	text	
stock_quantity	int	
price	decimal	
color	varchar	
size	int	
product_id	int	

accounts		
id	integer	NN
is_active	boolean	NN
created_at	timestamp	NN
updated_at	timestamp	NN
username	varchar	NN
email	varchar	NN
password	varchar	NN
account_type	AccountType E	

transaction_group

billing		
id	integer	NN
is_active	boolean	NN
created_at	timestamp	NN
updated_at	timestamp	NN
payment_method	PaymentMethod E	NN
address	varchar	NN
status	PaymentStatus E	NN
payment_at	timestamp	NN
amount	decimal	NN
transaction_id	int	

transaction_items		
id	integer	NN
is_active	boolean	NN
created_at	timestamp	NN
updated_at	timestamp	NN
quantity	int	NN
price	decimal	NN
product_model_id	int	NN
transaction_id	int	

transactions		
id	integer	NN
is_active	boolean	NN
created_at	timestamp	NN
updated_at	timestamp	NN
total_amount	decimal	NN
transaction_at	timestamp	NN
account_id	int	NN
billing_id	int	NN

Link ERD: https://dbdiagram.io/d/UDT-Test-672a1d32e9daa85aca6307c8

▼ Question 3:

1. With ERD you have already designed on question 1, what database are you using to implement?

   a. **Database Selection**

      The database selected for this project is **PostgreSQL**, known for its strong data integrity and scalability, which are essential for e-commerce platforms.

2. Why are you using that? What is the strong and weak point of it?

   Here is the reason why I decide to use PostgreSQL.

Strengths:

- **ACID Compliance**: PostgreSQL provides transaction support for data consistency across purchases and billing, vital in e-commerce.
- **JSON Support**: Facilitates flexibility for unstructured product data.
- **Strong Community and Extensibility**: Supported by a vast community, PostgreSQL offers numerous extensions and tuning options.

Weaknesses:

- **Memory Usage**: Compared to some NoSQL databases, PostgreSQL might require more resources.
- **Complexity**: Setup and tuning can be more complex, especially for distributed deployments.

3. Write docker-compose.yml to start the database locally. (1 Point) 4. Setup Spring (1 Point)

    a. **Docker setup**

        i. `docker-compose.yml`

```
version: '3.8'

services:
  db:
    image: postgres:latest
    container_name: ${CONTAINER_NAME}
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
    ports:
      - ${PORT}
    volumes:
      - postgres_data:/var/lib/postgresql/data

volumes:
  postgres_data:
```

ii. `.docker.env`

```
POSTGRES_USER=postgres
POSTGRES_PASSWORD=admin
POSTGRES_DB=udt_ecommerce
CONTAINER_NAME=udt_ecommerce_db
PORT=5432:5432
```

c. Command line for running the docker file:

```
docker-compose --env-file .docker.env up -d
```

4. Setup Spring

a. Setup file `application.properties`

```
spring.datasource.url=jdbc:postgresql://${DB_HOST}:$
spring.datasource.username=${DB_USERNAME}
spring.datasource.password=${DB_PASSWORD}
spring.jpa.properties.hibernate.dialect= org.hibernat

spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update
```

b. Setup file `.env`

```
DB_USERNAME=postgres
DB_PASSWORD=admin
DB_HOST=localhost
```

```
DB_PORT=5432
DB_NAME=udt_ecommerce
```

5. APIs based on this ERD would include:

   a. **Authentication API**

      - `POST /auth/login`: Authenticates a user and returns a token.
      - `POST /auth/logout`: Logs out the current user.
      - `POST /auth/register`: Registers a new customer account.

   b. **Customer API**

      - `GET /customers/{id}`: Retrieve customer profile by ID

   c. **Agency API**

      - `GET /agencies/{id}`: Retrieve agency profile by ID.
      - `GET /agencies/{id}/products`: List all products created by an agency.

   d. **Admin API:**

      - `GET /admin/agencies`: List all agencies.
      - `POST /admin/agencies`: Create a new agency.
      - `PATCH /admin/agencies/{id}`: Update agency information.
      - `DELETE /admin/agencies/{id}`: Delete an agency.
      - `GET /admin/customers`: List all customers.
      - `GET /admin/customers/{id}`: Retrieve a specific customer's information.
      - `GET /admin/transactions`: List all transactions.
      - `GET /admin/billing`: Retrieve all billing records.
      - `GET /admin/products`: List all products.

   e. **Product API:** ✅

      - `POST /products`: Create a new product (Agency only).
      - `GET /products` : List all available products across agencies.

- `GET /products/{id}`: Retrieve product details.
- `PATCH /products/{id}`: Update product details (check agency owner before update).
- `DELETE /products/{id}`: Delete a product (check agency owner before delete).

f. **Cart:**

- `GET /carts`: Retrieve cart details of current customer.
- `POST /carts`: Add an item to a specific cart.
- `PATCH /carts/items/{itemId}`: Update a cart item (e.g., quantity or status).
- `DELETE /carts/items/{itemId}`: Remove an item from the cart.

g. **Transaction API**

- `GET /transactions`: List all transactions (filter transactions by specific accounts).
- `GET /transactions/{id}`: Retrieve a specific transaction.
- `POST /transactions`: Create a new transaction (upon checkout).
- `GET /transactions/{id}/items`: Retrieve items associated with a transaction.

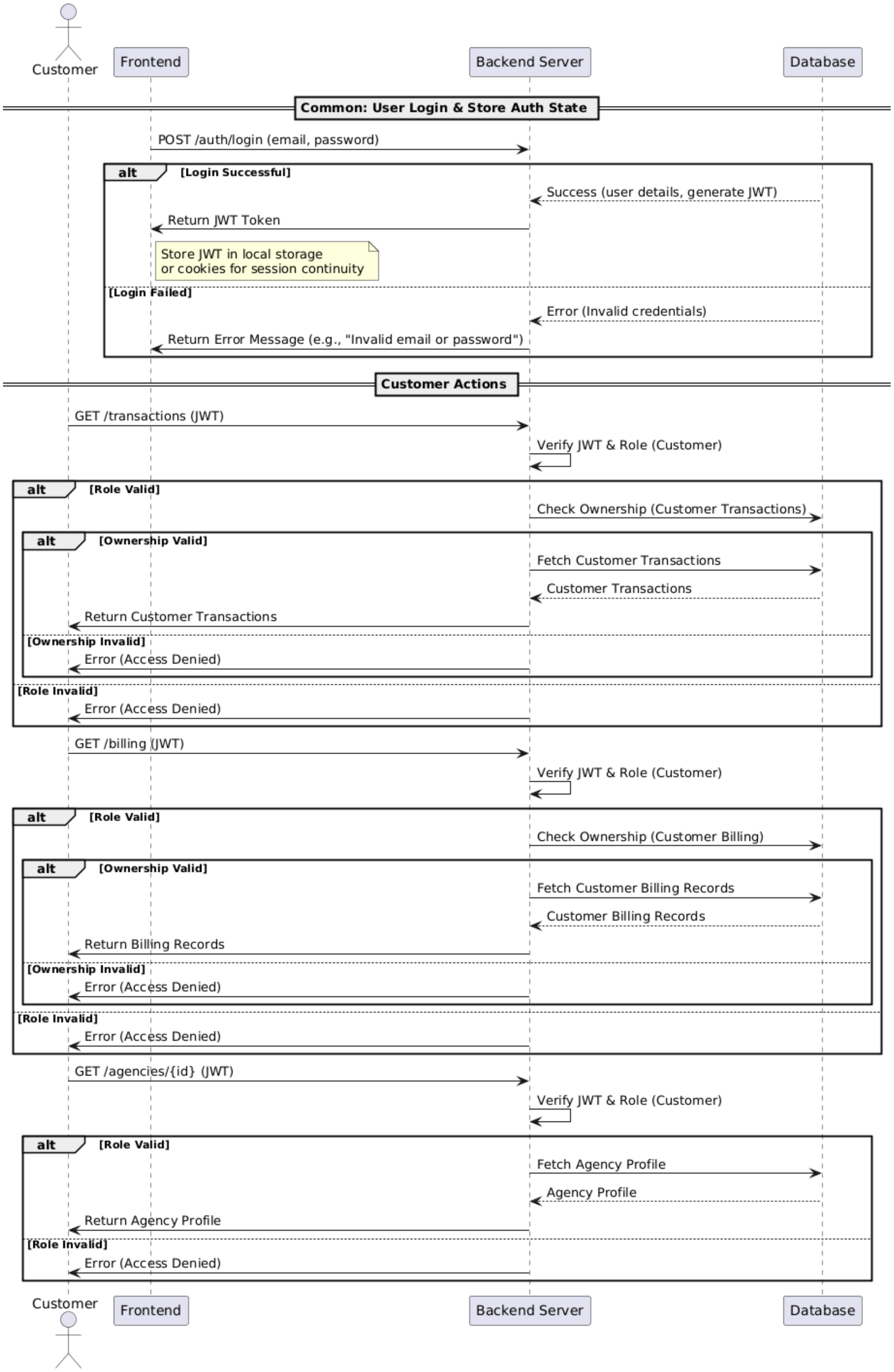
h. **Billing API:**

- `GET /billing`: List all billing records ((filter billing by specific accounts).
- `GET /billing/{id}`: Retrieve details of a specific billing record.
- `POST /billing`: Create a new billing record (on successful payment).
- `PATCH /billing/{id}`: Update billing details (e.g., payment status).
- `DELETE /billing/{id}`: Delete a billing record.

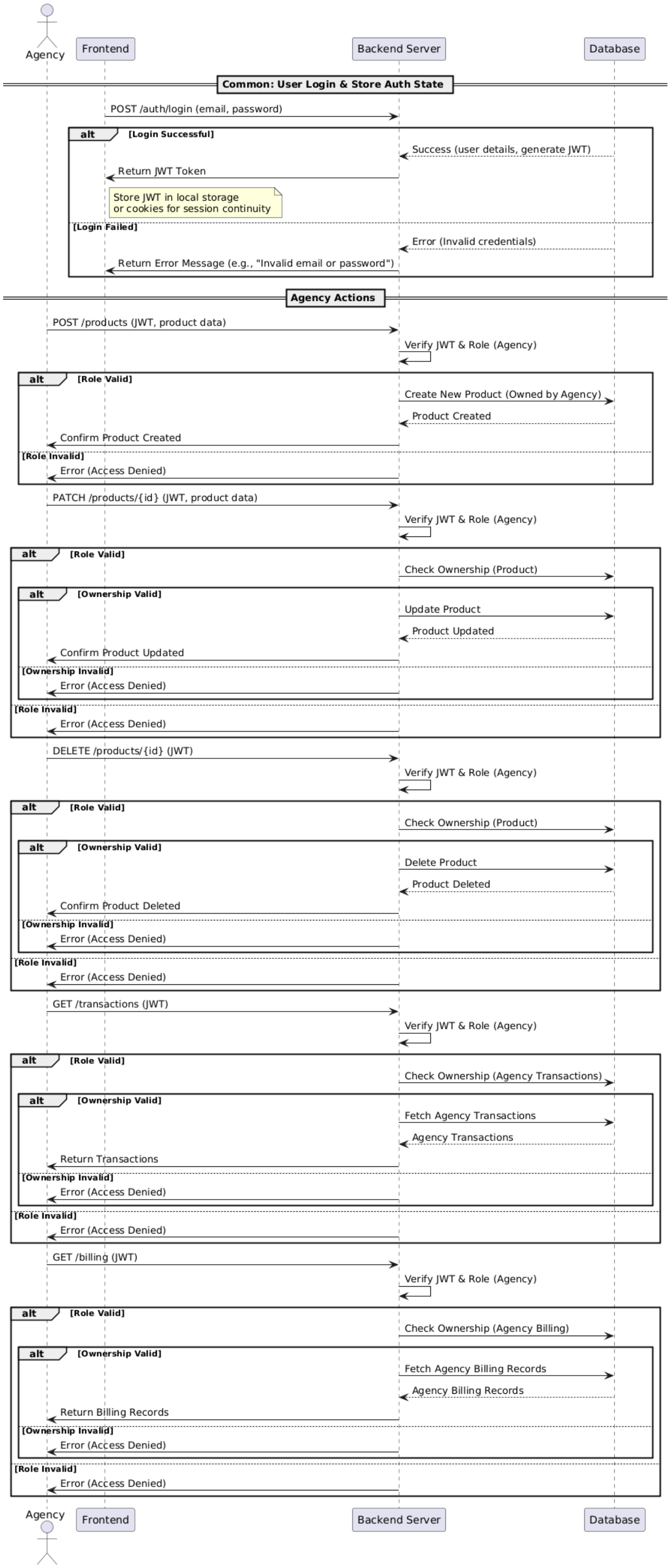
▼ Question 4:

▼ Write a sequence diagram to build a solution for authentication and authorize adapt the list of features below.

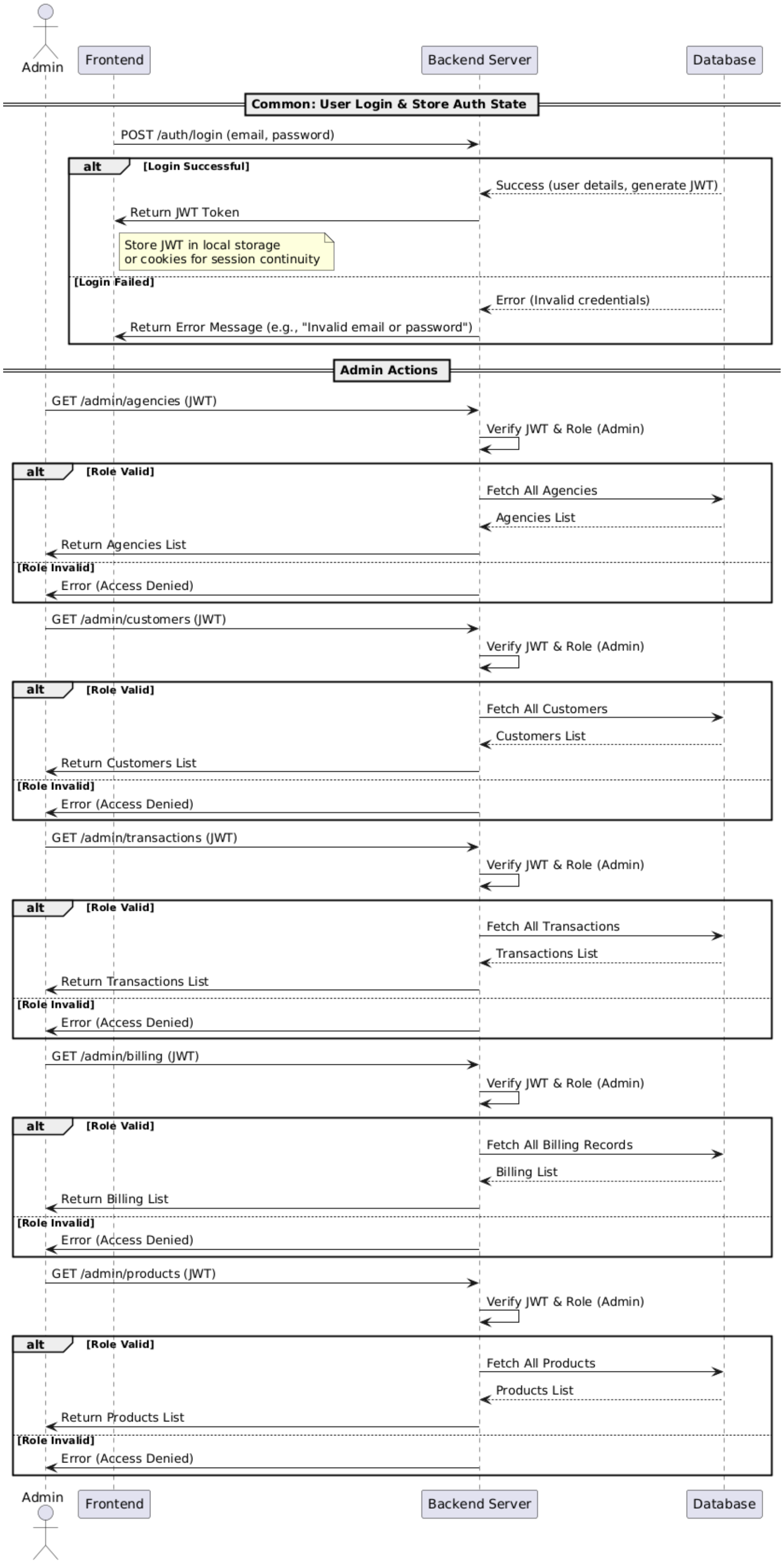
1. Customer: Can log in, log out. Read transaction, billing linked with product. Read agency information.



2. Agency: can read/create/update/delete of own product. Agency can read its own transactions, own billing.

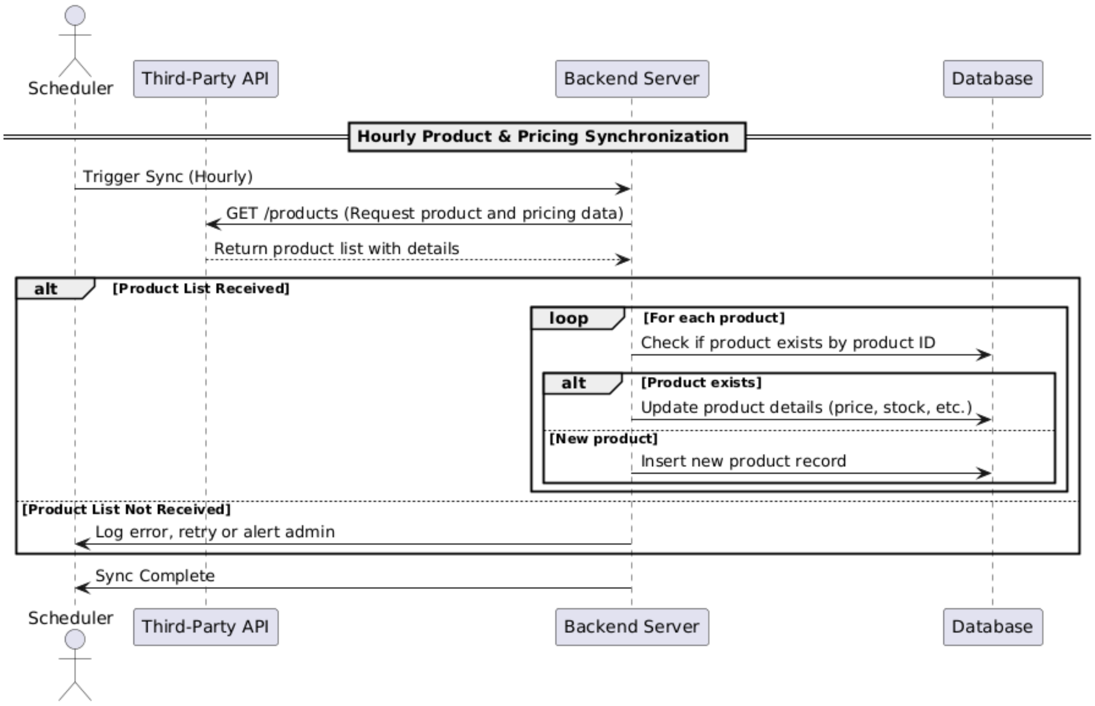

Agency
Frontend
Backend Server
Database
Common: User Login & Store Auth State
POST /auth/login (email, password)
alt
[Login Successful]
Success (user details, generate JWT)
Return JWT Token
Store JWT in local storage
or cookies for session continuity
[Login Failed]
Error (Invalid credentials)
Return Error Message (e.g., "Invalid email or password")
Agency Actions
POST /products (JWT, product data)
Verify JWT & Role (Agency)
alt
[Role Valid]
Create New Product (Owned by Agency)
Product Created
Confirm Product Created
[Role Invalid]
Error (Access Denied)
PATCH /products/{id} (JWT, product data)
Verify JWT & Role (Agency)
alt
[Role Valid]
Check Ownership (Product)
alt
[Ownership Valid]
Update Product
Product Updated
Confirm Product Updated
[Ownership Invalid]
Error (Access Denied)
[Role Invalid]
Error (Access Denied)
DELETE /products/{id} (JWT)
Verify JWT & Role (Agency)
alt
[Role Valid]
Check Ownership (Product)
alt
[Ownership Valid]
Delete Product
Product Deleted
Confirm Product Deleted
[Ownership Invalid]
Error (Access Denied)
[Role Invalid]
Error (Access Denied)
GET /transactions (JWT)
Verify JWT & Role (Agency)
alt
[Role Valid]
Check Ownership (Agency Transactions)
alt
[Ownership Valid]
Fetch Agency Transactions
Agency Transactions
Return Transactions
[Ownership Invalid]
Error (Access Denied)
[Role Invalid]
Error (Access Denied)
GET /billing (JWT)
Verify JWT & Role (Agency)
alt
[Role Valid]
Check Ownership (Agency Billing)
alt
[Ownership Valid]
Fetch Agency Billing Records
Agency Billing Records
Return Billing Records
[Ownership Invalid]
Error (Access Denied)
[Role Invalid]
Error (Access Denied)
Agency
Frontend
Backend Server
Database

3. Read billing, transaction, product, customer, agency.


Admin
Frontend
Backend Server
Database
Common: User Login & Store Auth State
POST /auth/login (email, password)
alt
[Login Successful]
Success (user details, generate JWT)
Return JWT Token
Store JWT in local storage
or cookies for session continuity
[Login Failed]
Error (Invalid credentials)
Return Error Message (e.g., "Invalid email or password")
Admin Actions
GET /admin/agencies (JWT)
Verify JWT & Role (Admin)
alt
[Role Valid]
Fetch All Agencies
Agencies List
Return Agencies List
[Role Invalid]
Error (Access Denied)
GET /admin/customers (JWT)
Verify JWT & Role (Admin)
alt
[Role Valid]
Fetch All Customers
Customers List
Return Customers List
[Role Invalid]
Error (Access Denied)
GET /admin/transactions (JWT)
Verify JWT & Role (Admin)
alt
[Role Valid]
Fetch All Transactions
Transactions List
Return Transactions List
[Role Invalid]
Error (Access Denied)
GET /admin/billing (JWT)
Verify JWT & Role (Admin)
alt
[Role Valid]
Fetch All Billing Records
Billing List
Return Billing List
[Role Invalid]
Error (Access Denied)
GET /admin/products (JWT)
Verify JWT & Role (Admin)
alt
[Role Valid]
Fetch All Products
Products List
Return Products List
[Role Invalid]
Error (Access Denied)
Admin
Frontend
Backend Server
Database

▼ Question 5: Good job, right now our application needs to synchronize products, pricing of the Agency by using third-party API.

▼ 1. Write a sequence diagram to build a solution to save, merge products data from third-party API to our database. (Third-party API data change every hour) :


Scheduler
Third-Party API
Backend Server
Database
Hourly Product & Pricing Synchronization
Trigger Sync (Hourly)
GET /products (Request product and pricing data)
Return product list with details
alt
[Product List Received]
loop
[For each product]
Check if product exists by product ID
alt
[Product exists]
Update product details (price, stock, etc.)
[New product]
Insert new product record
[Product List Not Received]
Log error, retry or alert admin
Sync Complete
Scheduler
Third-Party API
Backend Server
Database

▼ 2. What are the strong and weak points of your solution? How to improve that?

Strengths:

- **Clear Flow:** The sequence of operations from the scheduler triggering sync to the server updating the database is easy to follow.
- **Error Handling:** If the product list isn't received, an error is logged, and admin is alerted.
- **Granularity:** Each product is checked and updated individually, ensuring data consistency.

- **Modular:** Each component (Scheduler, Server, API, Database) has a clear role, making the system easy to maintain.

Weaknesses

- **No API Error Handling:** If the third-party API fails or returns an error, there's no fallback plan.
    - **Fix:** Add an error handling flow for API failures (retry, alert, log).
- **Concurrency Issues:** If multiple processes update the same product at once, there could be conflicts.
    - **Fix:** Use database locking or transactions to prevent conflicts.
- **Rate Limiting:** If the API has rate limits, hitting them could cause issues.
    - **Fix:** Implement rate-limiting or queuing requests.
- **Product Update Logic:** The check only compares product IDs, which may miss updates to pricing or stock.
    - **Fix:** Compare the full product details (price, stock) before updating.
- **No Logging for Updates:** There's no logging for product updates or insertions.
    - **Fix:** Log every product update or insert for audit purposes.
- **Performance with Large Lists:** Handling large product lists in a loop might slow down the process.
    - **Fix:** Use batch processing or queueing for large sets of products.

Improvement

- **Add API Error Handling:** If the third-party API fails or returns an error, the system should log the error and alert the admin. A retry mechanism could also be implemented for transient errors.
- **Add Database Locking:** To prevent concurrency issues, especially when multiple processes might update the same product

simultaneously, database locking or transaction mechanisms should be used to ensure data consistency.

- **Implement Rate-Limiting:** If the third-party API has rate limits, the system should respect these limits. Implementing a queuing mechanism or waiting before retrying requests can help avoid hitting these limits.
- **Compare Full Product Details:** When updating products, it's important to not just check for product existence by ID, but also compare details like price and stock before deciding to update or insert. This will prevent unnecessary updates.
- **Log Product Changes:** Every product update or insertion should be logged for audit purposes, so that any changes can be tracked and traced back if needed.
- **Use Batch Processing:** If the product list is very large, processing all products one by one could slow down the system. Batch processing or queuing can help handle large sets of products more efficiently, reducing processing time.