Name: Trần Quang Minh

TYME HOMEWORK

Challenge 1

Your team member is going to expose these APIs, what is your feedback regarding to API design?

```
Get /users
POST /users/new
POST /users/:id/update
POST /users/:id/rename
POST /users/:id/update-timezone
DELETE /users/delete?id=:id
```

Optional: what's your personal opinion on CRUD endpoints?

Answer:

1. GET /users

The GET API that returns all users, it's not a good idea in a real scenario with many users. We should use pagination, limits, and sorting through query parameters.

My suggestion:

GET /users?page={page_number}&limit={limit_number}&sort={field_name}&order={direction}

Request parameters:
- **page:** for pagination.
- **limit:** to limit the number of users returned.
- **sort:** to define sorting criteria.
- **order:** to specify the sort direction (ASC or DESC).

2. POST /users/new

This is an endpoint to create a new user, but it does not follow Restful API design principles. Including "new" in the URL is redundant because the POST method already implies the creation of a new resource.

My suggestion: ***POST /users***

3. POST /users/:id/update

This API does not follow Restful API design principles. To update a user by ID, we should use the PUT method and modify the endpoint to align with Restful API design.

My suggestion: ***PUT /users/:id***

4. POST /users/:id/rename

This API does not follow the Restful API design. We avoid using the verb '**rename**.' This API is used to update a user's name, so we should use PATCH /users/:id with the request body: {"name": "new name"} to change the name.

My suggestion:

PUT /users/:id
Request body: {"name": "new name" }

5. POST /users/:id/update-timezone

This API does not follow the Restful API design.
We should use the PATCH method instead of POST because we only want to perform a partial update of a resource. Additionally, we should update 'update-timezone' to 'timezone.

My suggestion:

PATCH /users/:id
Request body: {"timezone": "New Time Zone" }

6. DELETE /users/delete?id=:id

This endpoint is used to delete a specific user, but it does not follow Restful API design principles.
We should avoid using the verb 'delete' and change the ID from a request parameter to a path parameter.

My suggestion: ***DELETE /users/:id***

What's your personal opinion on CRUD endpoints?

In my opinions, I have some standards when building CRUD endpoints:

1. HTTP Method

- **GET:** Retrieve data from a server. It should be idempotent.
- **POST:** Send data to a server to create a new resource.
- **PUT:** Update an existing resource. It should be idempotent.
- **PATCH:** Used to update a resource partially. It should be idempotent.
- **DELETE:** Remove a specified resource. It should be idempotent.

2. URL

- URLs should use nouns rather than verbs for resources. For example, **/users** instead of **/getUsers**.
- Use plural nouns for collections of resources. For example, **/users** for a collection of user resources.
- Use consistent naming conventions, such as lowercase letters and hyphens to separate words. For example, **/user-profiles** instead of **/userProfiles**.
- Include versioning in the URL to manage changes to the API. For example, **/v1/users** for version 1 of the users resource.

3. Path Parameter, Query Parameter, Request Body

- **Path parameters** are used to identify specific resources in the URL path.
Example: GET /users/{*userId*} where {*userId*} is a path parameter.
- **Query parameters** are used to filter, sort, and paginate resources. They are appended to the URL after a "?".
Example:
    - /users?sort=age&limit=10
    - /users?sort=age&limit=10&page=2
- **Request Body** is used to send data to the server for creating, updating resources.

4. HTTP Status Code

Send the right HTTP Status Code in response. For example:

- 200 OK for success **GET** request**.**
- 201 Created for **POST** request**.**
- 204 No Content for **DELETE** request**.**

To handle errors effectively, we need to implement exception handling that assigns the appropriate error codes to different exceptions. For example:

- 404 Not Found for EntityNotFoundException
- 400 Bad Request for IllegalArgumentException
- 403 Forbidden for authentication failures

And with server error we can use **5xx: Server Errors**. For example:

- **500 Internal Server Error** for the server encountered an unexpected condition.
- **501 Not Implemented for** the server does not support the functionality.

5. API definition

To ensure clarity and ease of use, we should document to describe **URL**, **HTTP Method**, **Sample Request**, **Sample Response** for success case and error case also.

Challenge 2

What is your feedback when you review this code? Would you give this a `lgtm`?

```
public class Account {
  private int debitBalance;
  private int creditBalance;

  void increaseBalance(int amount, boolean isCredit){
    if(isCredit) {
      this.creditBalance += amount;
    } else {
      this.debitBalance += amount;
    }
  }
  //getter,setter
}
```

Answer:

I would **NOT** give “Looks good to me” for this code, because there are a few points I request change:

- Using an **int** for monetary values can lead to incorrect results. Therefore, we must use the **BigDecimal** data type for handling money.
- Instead of using a boolean flag like **isCredit** to switch between actions, we should create distinct methods such as **debitBalance** and **creditBalance**.
- If we prefer to use a single method, consider using enums like **BalanceType.CREDIT** and **BalanceType.DEBIT** as parameters. This approach is clearer than using a boolean flag.

- Ensure that the amount is validated to be a suitable value before adding it to the balance.
- Format the code properly and generate **Javadoc** documentation for all public methods.

Challenge 3

What performance issue related to the database can you see in the following code, and how are you going to fix it?

```
public void loadStudents(){
  List<Course> courses = loadFirst1000Courses();
  for (Course course: courses) {
    List<Student> student = loadStudentFromCourse(course);
    print(student);
```

Answer:

There are two issues:

- Performance issue: The current approach retrieves 1000 courses and then fetches the list of students for each course, resulting in at least 1000 database hits. This is highly inefficient and leads to poor performance.

- Duplication issue: In a real-world scenario, the *student-course* relationship is *many-to-many*. This means students can be enrolled in multiple courses simultaneously. So, the print results will contain many duplicate entries for students.

Solution:

- Reduce Number of Courses: Review the current logic and consider using batch size queries to limit the number of courses retrieved at one time.

- Reduce Database Hits and Data Fetching: Currently, we retrieve 1000 courses and then fetch the list of students for each course, resulting in 1000 database hits. Additionally, we fetch all data for both course and student objects, which may not be necessary.

I think we should review the logic of **loadStudentFromCourse(course)** to determine the exact data needed from the course to get the student list. If only **course_id** is required, we can optimize with two queries:

```
SELECT c.id FROM courses c LIMIT 1000;
SELECT s.* FROM students WHERE s.course_id IN (list of course ids);
```

- Parallel Processing:
Review the print(student) method. If it contains extensive logic, consider using parallel processing to reduce the time taken to process the student list. For example:

```
students.parallelStream().forEach(student -> { print(student); });
```

Challenge 4

This piece of code is the logic behind an API for transfer money, we're using RDBMS e.g. MySQL as our persistent layer. What database-related (2-3) issues would you find in this method?

```
function transfer(fromAccId, toAccId, amount) {
  var from = findByAccountId(fromAccId);
  var to = findByAccountId(toAccId);
  if(from.balance < amount) {
    throw new BalanceErrorException("not enough balance to transfer");
  }
  updateBalance(from, balance - amount);
  updateBalance(to, balance - amount);
}
```

Answer:

There are some issue in this code:

Concurrent Modifications: This function can be called from multiple threads, leading to inconsistent balances for **fromAcc** and **toAcc**.

Example: **fromAccount** has balance of 100$. And we have 2 threads calling to transfer in parallel. First thread transfer 80$ and second thread transfer 50$. Obviously, the balance of **fromAccount** can not enough to transfer 2 these transactions. But 2 thread is transferring in parallel, so **fromAccount** 's balance has not been updated of one of thread. It can pass the code if **(from.balance < amount)** → **return false** and go to **updateBalance**. It is issue.

Missing transaction: We have to create transaction for this method. Because we need to make sure that after amount is deducted from **fromAccount**, it must be added to **toAccount**.

Missing Account Availability Validation: We need to check if the **fromAccount** and **toAccount** are available (i.e., the entities exist in the database and their status is available). If not, we should throw an **EntityNotFoundException** or a similar exception.

To fix the issues in the code, we can implement the following solutions:

Transaction Management: Use JDBC transactions, JPA, or the **@Transactional** annotation if using the Spring framework to ensure atomicity and consistency.

Ensure Thread Safety: Use synchronized access or concurrent locks from **java.util.concurrent** to prevent concurrent modifications.

Pessimistic Locking: Apply pessimistic locking at the database level to prevent other transactions from modifying the data until the current transaction is complete.

Exception Handling: Handle exceptions such as entity not found, insufficient funds, and database exceptions. Ensure that the transaction can be rolled back if an exception occurs.

Logging: Since transferring money is a critical function, add logging for each step to facilitate investigation in case of unexpected issues.

Challenge 5

We found this code in our git repository, what problems related to application security can you see in the following code?

```
// A service store user credentials and authenticate user (login)
class User {
  /**
   * Use salted password
   */
  private static final String PASSWORD_SALT = "tymesalt";

  /**
   * Stored password hashed
   */
  private String hashedPassword;

  /**
   * Authenticates user against given password, return true if the password matches
   * @param password
   * @return
   */
  public boolean verifyPassword(String password){
    if (isNullOrEmpty(hashed_password) || isNullOrEmpty(password)) {
      return false;
    } else {
      return this.hashPassword(password).equals(hashedPassword);
    }
  }

  /**
   * Update password
   * @param newPassword
   */
  public void changePassword(String newPassword) {
    this.hashedPassword = this.hashPassword(newPassword);
  }

  /**
   * Generate hash from given password
   * @param password
   * @return
   */
  private String hashPassword(String password){
    //generate md5hash of a string
    return md5Hash(password + PASSWORD_SALT);
  }
}
```

Answer:

There are several security issues in this code:

- **Static Salt Usage:** Using a static salt means the same password will always generate the same hash for all users with the same plain text password, making it vulnerable to rainbow table attacks. To improve security, use a dynamic salt.

- **Storing Passwords as Strings:** Storing passwords as strings in the application is insecure because strings are immutable and stored in the string pool, which cannot be cleared until garbage collection occurs. This can lead to password leaks. Instead, store passwords securely in the database.

- **Insecure Hasing Algorithm (MD5):** MD5 is not secure for hashing passwords because it is fast and easy to implement. Use more secure hashing algorithms such as bcrypt or Argon2.

- **Missing Input Validation:** Implement password complexity rules to ensure strong passwords, such as minimum length, inclusion of special characters, and preventing reuse of recent passwords.

Challenge 6

Database queries are getting slow when the database size increases. What are your suggestions to improve performance over the time?

Answer:

There are several ways to improve performance:

- **Indexing:** Ensure that columns used in **WHERE, IN**, **ORDER BY**, and **GROUP BY** clauses have appropriate indexes. Use the execution plan to verify that indexes are being utilized effectively. Additionally, we should remove any unused indexes to avoid potential performance issues.

To check unused or less used index:

In MySQL: Use the **INFORMATION_SCHEMA** tables to identify unused indexes.

In PostgreSQL: Use **pg_stat_user_indexes** view provides statistics about index usage.

- **Optimize Queries:** Use the execution plan to understand and improve query performance. For example, remove unused fields in **SELECT** statements to reduce fetching time, be cautious with complex **JOIN** operations, avoid running queries inside loops.

- **Caching:** Implement caching for frequently accessed data to minimize database hits and improve response times.

- **Partitioning:** Divide large databases into smaller partitions to reduce the amount of data scanned during queries, thereby improving performance.

- **Consider Using a Search Engine or Data Warehouse:** For specific business logic such as daily reports and analysis, consider offloading these tasks to a data warehouse (e.g., Amazon Redshift, Google BigQuery, IBM Db2) or using a search engine like Elasticsearch to improve query performance and reduce the load on the main database.

Challenge 7

System design

Design a high load payment system.
The problems:
- During the Black Friday, the system needs to handle 3000 concurrent requests.
- The frequency of balance/transaction inquiry is usually 5 times higher than requesting payments e.g. 2500 reads + 500 writes
- Reliability is expected, e.g. we don't want to approve any inappropriate payments.
- User experience is also important, low-latency is expected.

Resource limit:
- We have 6 servers where up to 3 servers could be used for the relational databases. One database can open 250 connections.

Answer:

1. Functional requirements:
- Get balance
- Get transactions
- Make a payment

2. **Non-functional requirements:**

- Concurrent request: need to handle 3000 concurrent requests during Black Friday
- Read heavy: 2500 reads & 500 writes
- Low-latency: provide low-latency user experience
- Reliability: minimize the risk of inappropriate payments
- Availability: maintain system uptime and minimize downtime
- Fault-tolerance: ensure the system can withstand failures and continue operating

3. API definitions:

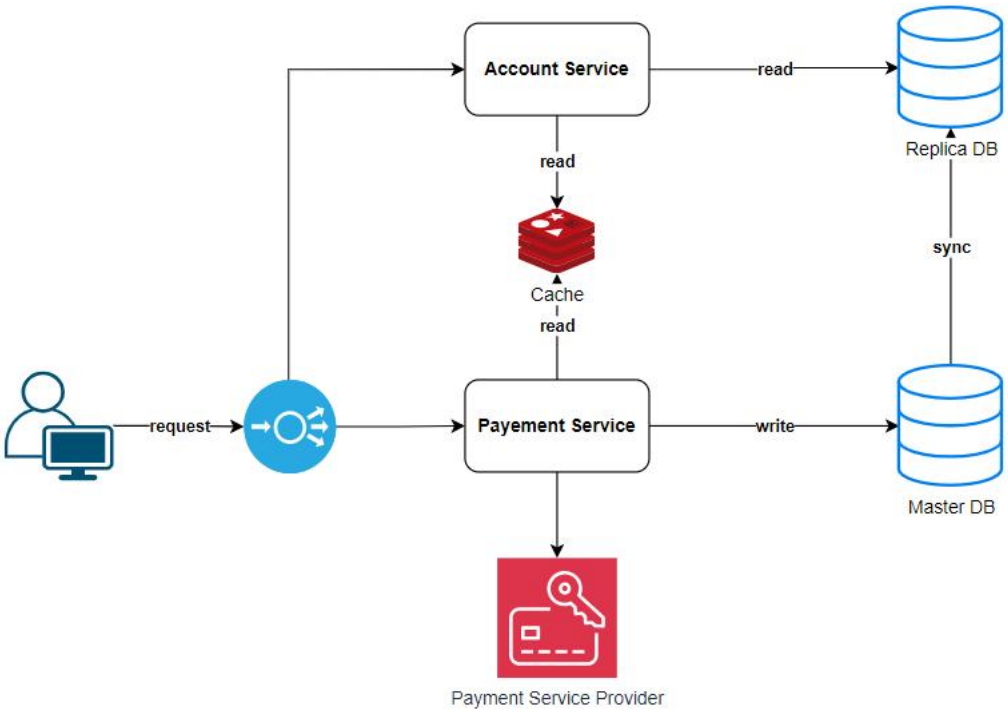
GET /accounts/:id/balance: this endpoint to get balance of specific account.
- **Path Parameter:** id is used to identify account.

GET /accounts/:id/transactions: this endpoint to get list transactions of specific account.
- **Path Parameter:** id is used to identify account.
- **Query Parameter:**
  - **status:** is used to filter transactions by status, we can filter multi statuses, separated by comma.
  - **page:** for pagination.
  - **sort:** to define sorting criteria.
  - **order:** to specify the sort direction (ASC or DESC).

POST /payment: this endpoint to execute a payment process.
- **Request body:** should be included user_id, amount, currency, payment_details like card_number, expiry_date, cvv, …

GET /payment/:id: this endpoint to get payment details

4. System Architecture



There are list components in my design:

Client: The application or user interface that users interact with to access the payment system.

Load Balancer: Distributes incoming traffic across multiple back-end service instances to ensure scalability and availability.

Payment Service: Handles requests to process payments, including necessary approvals and validations.

Account Service: Handles requests to view balances, retrieve transactions.

Master Database: The primary database server that manages the master copy of all data.

Replica Database: Replicated database servers that maintain copies of the data for redundancy and improved read scalability.

Redis Cache: Enhances performance by caching frequently accessed data, such as user balances and transaction records.

PSP (Payment Service Provider): External service providers that process payments, including PayPal, Stripe, Visa processors, or other payment gateways.