Assignment T2: Revised Project Proposal

Team Sangria

Zhiyuan Lin(zl2989), Zoe Cui(qc2292), Linyu Li(ll3465), Tianzhi Huang(th2888)

Part 1:

IA: Asif Mallik

Scheduled Meeting time: 10/23/2021 23:00:00 via discord

Suggestions from IA:

1. Our original service is too specific as a winemaking subsystem for a game. We should make it more general.

   We modify the service so that it can take different recipes and different item lists to customize the ecosystem for a game. By doing this our service can be used for different games as long as they need an inventory system. We think that our system will be especially useful/realistic to use for companies who'd like to develop multiple games that have recipe and inventory components (since network traffic and delay can be minimized).

2. The broker mediated request-response mechanism is not the core of this task. Should put more effort on other parts.

   Yes, we will focus on the core functionality of our services, and we will only build the broker mediated request-response system if time allows.

3. Original proposal contains few details of testing.

   We included more details in this revised version.

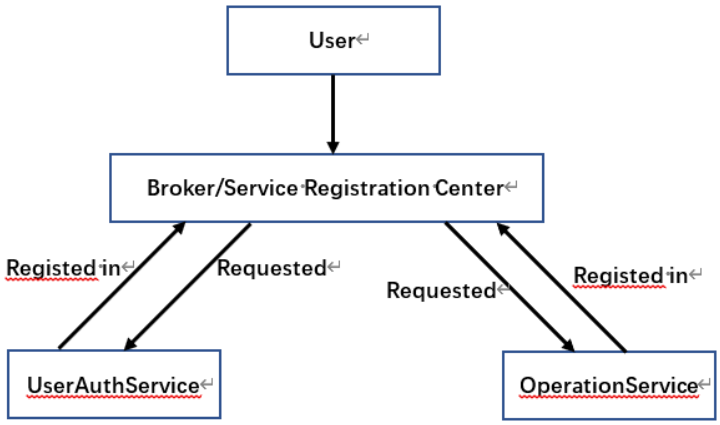
Part 2:

Our service serves as inventory management for a game. A game can call our service and login as a game manager to manage game item lists, recipe lists and its own user lists. A player can log into their own account and have their own game progress and inventory in corresponding games. A Player can get the materials through some means (for example, like in an incremental game, players acquire new items as time passes). The database records recipe lists and item lists for different games and if the player provides a correct combination of materials that appear

in one recipe, they can successfully make an item. The database also stores a player's inventory. Each item has some attributes, for example, price, effect, etc.

We are also planning on using a broker mediated request-response to implement our service. This means that we have two backend services, one is for user authentication and the other one is for wine making functionalities. The reason why we choose this service architecture is that we want to challenge ourselves to try something other than traditional client-server architecture. Moreover, the broker architecture is more suitable if our backend services expand to support other functionalities like battle, gacha system, and trading.

The overall architecture of our inventory management backend service will be like:



- User:
  - Hypothetical game is the user of our service
- Broker/Service Registration Center:
  - In order to simulate the situation in industry where our two services could be possibly deployed in different machines and their machine's IP address is not static, we plan to use Service Registration Center (Eureka or zookeeper) to register and manage services. We will deploy the Service Registration Center at machine 1 with IP xxx.xx.xx.xx1, UserAuthService at machine 2 with IP xxx.xx.xx.xx2, and OperationService at machine 3 with IP xxx.xx.xx.xx3. When we start the services, they will first look for the Service Registration Center at xxx.xx.xx.xx1 on port 8091, register themselves as service providers with their corresponding IP and port.
- UserAuthService:
  - UserAuthService will provide services allowing for new user registration, user login, user authentication check, user character's information display, and any other functionalities related to user identity. Following are the tentative API for the service:
    - UserAuthService API table:

Request API	Request Method	Parameters	Remarks

public_ip/UserAuth/register	POST	Username, password, gameID	New user/game manager registration
public_ip/UserAuth/login	POST	Username, password	User/manager log in, return token
public_ip/UserAuth/info	GET	token	Get user/manager information
public_ip/UserAuth/logoff	POST	token	User/manager log off
public_ip/UserAuth/delete	POST	token	Delete user account

        - We are considering adding a required additional parameter <sign> to be some form of encryption (maybe MD5) to every request to prevent unauthorized requests.
- OperationService:
    - Our user, aka a game, also talks to operationService and passes players of the game to it. OperationService will provide services for various player operations during the process of game, which include acquiring materials, using items, selling items, view inventory, view recipes, drop items, and so on. Each user (game), also has an admin who we identify as the manager of the game and have a set of privileged apis like adding/deleting/modifying recipes/items to the game recipe/item list. Following are the tentative API for the service:
        - OperationService API table:

Request API	Request Method	Parameters	Remarks
public_ip/operation/explore	POST	token	Getting materials randomly
public_ip/operation/bag	GET	token	View all items in the user's inventory
public_ip/operation/recipes	GET	token	View recipes list of a game
public_ip/operation/useItem	POST	itemId, token	Use certain item to gain special effect
public_ip/operation/viewItem	GET	itemID, token	View certain item information
public_ip/operation/sellItem	POST	itemID, token	Sell a item to get coin
public_ip/operation/dropItem	POST	itemID, token	Drop off a useless item
public_ip/operation/makeItem	POST	[itemID1,itemID2...], token	Try to make an item with specified items
public_ip/operation/insertItem	POST	itemId, attributes, token	Add an item to the item list. Can only be used by the manager.
public_ip/operation/deleteItem	POST	itemId, token	Delete an item from the item list. Can only be used by the manager.
public_ip/operation/insertRecipe	POST	recipeId, [itemID1,itemID2...], generatedItemId, token	Add a recipe to the recipe list. Can only be used by the manager.
public_ip/operation/deleteRecipe	POST	recipeId, token	Delete a recipe from the recipe list. Can

			only be used by the manager.

- Similar to UserAuthService, we are planning to add a required additional parameter <sign> to be some form of encryption/hash (maybe MD5) to every request to prevent unauthorized requests.

Our responses for the three questions listed in the Criteria:

1. Is there meaningful data that persists across executions?

Our meaningful data, including game inventory list, game info, recipes, user inventories, etc, is persistent and stored in the database.

2. Does the service support multiple clients, distinguish clients from each other, and protect client data from other clients?

Yes, different clients will be differentiated by the parameter <token>(which is assigned and returned by the login method). Token and user will be a one-to-one mapping relationship and token will also be used to identify users from different games.

3. Is the service realistically useful for multiple different apps, not just the backend of a single app?

The service we designed could be used by any client as long as they have the similar context and data available as the parameters for our service. For example, it can be used for another game, which also needs an item inventory and all those functions above by incorporating our service with its own item inventory and recipe system.

Part 3:

During the development stage, we develop Java unit test cases for each function of the service to ensure that everything works as designed on the function level. When we finish developing and running unit test cases, we will perform the integrated test by designing specific use cases of different scenarios, including the happy path, alternative flow, and exception flow, in order to ensure that all desired functionalities could be realized across services. Parameter verification will be done at the beginning of each function, which checks whether this specific parameter combination is valid, and those test parameters will be included in the unit test cases.

More specifically:

- Unit Test:

    - For each controller and service that we implement, we will write a unit test file XXXController/ServiceTest.java under the directory of <project-directory>/src/test/ to perform white box test. We test each controller or service by running its corresponding XXXtest.java to determine the correctness of the implementation. Besides, we will use some third party tools to check the coverage of the test script we write to ensure maximum code coverage.
- Integrated Test:
    - Functionalities:
        - The integrated test of functionalities' correctness will be done by designing specific use cases and test cases of different scenarios, including the happy path, alternative flow, and exception flow, to check the correctness of the HTTP return values. This will be completed by the black box manner and we may use softwares like Postman or our browser debugging mode to help us.
    - Performance:
        - In order to assess the time performance of HTTP response and database I/O rate, we plan to use software like Jmeter to conduct the performance test under the situation of certain level of concurrency (several clients sending several requests at the same time), as well as preventing undesired behaviors and data corruption from happening when involving concurrency over clients and requests.

Our responses to the three questions listed in the criteria:

1. Testing for whether the meaningful data indeed persists across executions?

We will do some operations and then check whether the data is persistent. For example, We may add and delete items to the bag of one player, and check whether the items of another player is unaffected. We can also execute makeItem operations, and then check if the raw materials are removed from the inventory of the player, and the newly made item is added to it. Finally, we are also going to test when one recipe is added to one game, the data of other games are not affected.

2. Testing with API calls constructed to look like they come from different clients (e.g. different client IDs authenticated with password or key)?

We will use different tokens to represent different managers and players from different games to simulate the different clients.

3. Does the test plan demonstrate understanding of API testing, without relying on an interactive user

We will use Postman to call our apis and get running results which will be used to check whether our apis are logically right , and it does not rely on any interactive user.

Part 4:

Programming language: Java

Deployment machine: Google Cloud VM

Project framework: Springboot/SpringCloud

Database: MySQL/PostgreSQL

Coding IDEA: Eclipse, IntelliJ

Software & Tools: Postman, Maven, Jmeter, Java code Guidelines(style checker), JUNIT(static bug finder, coverage tracker, test runner will all be included as built-in features in the IDEA we use.)