Project 2: Online Marketplace Project Using Akka

Introduction

In this project you will reimplement only the Marketplace microservice in Akka. For the other two microservices (i.e., Account and Wallet), you will simply reuse your previous Docker+Spring based microservices. Account and Wallet should listen at http://localhost:8080 and http://localhost:8082, respectively. They will be launched via "docker run" directly (no need to use kubernetes in this project). Marketplace should listen on http://localhost:8081. This will be a single-node Akka program ~~(i.e., no cluster)~~. The request and response formats for Marketplace should be exactly the same as it was with the Spring project, so that all our Project 1 test cases remain valid. For simplicity, you can assume that no messages will ever be lost in the Akka implementation.

Structure of your Marketplace code

Since this project will need HTTP interaction, it won't be a pure Akka program. You would need a HTTP server that passes on the requests received by it to Akka actors. There are several ways to wrap a HTTP server around an Akka program. In this section, I suggest one recommended way, which is to use the Java library class `HttpServer`. Towards the end of this document, I give other suggestions also, which you could go with if you prefer. In all cases the primary actors (other than the root actor) will be the same and their structure and functionality will not really change. It is only the Http server <-> Actors linkage code that will change.

Regarding the structure and functionality of the main() function, the root actor, and the handle() method to pass to the HttpServer object, please see the skeleton code in this demo project.

Some references to learn about HttpServer:

- https://dzone.com/articles/simple-http-server-in-java
- https://stackoverflow.com/questions/3732109/simple-http-server-in-java-using-only-java-se-api. Search for the answer that contains the string "Allows multiple request serving via multiple threads using executor service."

Note that other than the sole use of ask() in the handle() method, there should be no other uses of ask() anywhere else in your code. Only tell() should be used.

The Product and Order actors

You will have an actor class for Products and an actor class for Orders. That is, each product and each order will be an actor. In particular, each product and each actor should be a *cluster-shared entity*. The entity ID of a product (resp. order) will be the corresponding product (resp. order) ID itself. The formats of the messages to these actors and the response messages from these actors is up to you. But I do have some further suggestions below.

The Gateway actor

This will be a singleton actor (i.e., using the Gateway class we will spawn only one actor). The demo project has a stub version of the Gateway class, which you need to flesh out. The Gateway actor should first, in its constructor, load each product mentioned in the csv file into a *cluster-sharded entity*.

The Gateway actor should have an incoming message type and an event handler corresponding to each request that the Marketplace service is supposed to handle. For the sake of simplicity, you can *omit* the following end-points from the Akka implementation of Marketplace:

- GET /products
- GET /orders/users/{userId}
- DELETE /marketplace/users/{userId}
- DELETE /marketplace

We now discuss the end-points that you do need to handle.

For GET /products/{productId}, simply pass on the request message to the corresponding product actor, and let that actor respond directly to the reply-to actor in the message.

For GET /orders/{orderId}, simply pass on the request message to the corresponding order actor, and let that actor respond directly to the reply-to actor in the message.

For PUT /orders/{orderId}, simply pass on the request message to the corresponding order actor, and let that actor respond directly to the reply-to actor in the message.

For POST /orders, the Gateway event handler should spawn a PostOrder actor, and pass on the request to this actor via its constructor. PostOrder will basically be a worker actor, somewhat similar to the Transaction actor in the banking example. PostOrder will communicate with the Account service, the Wallet service, and the product entities involved in the order, and will carry out the order creation following the requirements mentioned in the original Project 1 requirements doc (while being able to handle concurrent requests). The PostOrder actor will create the order actor in the end if all goes well. It will also respond back

directly to the reply-to actor in the message received by the Gateway actor. The design of this PostOrder actor will be one of the major tasks in this entire project.

For DELETE /orders/{orderId} also, you should have a worker actor of class DeleteOrder, which will do all the work related to canceling an order. The idea is similar to above.

NOTE: In order to use sharded entities, even though Marketplace will be a single-node Akka program, it must be started as being part of a cluster (i.e., as a single-node cluster). In order to enable akka cluster, you need certain elements in the `pom.xml` and you need an `application.conf` file in the `src/main/resources` folder. You also need specific code fragments in the `startup()` function and in the root actor to enable cluster and cluster sharding. See the `akka-raghavan-sharding-noPersistence-demo` project on the class web page as a sample (but change the port number to 8081 rather than 25251, and also update all functionality to suit this project's requirements).

Some suggestions about how you should do this project

Before you start writing code, do a detailed design on paper of the messages, the flow of messages under different scenarios, and the action to be performed upon the receipt of each message. Analyze these flows manually under different scenarios very carefully to make sure that your design will work. In message-passing programs it is easy to commit errors due to not understanding properly the delays that can happen in message delivery or even in spawning an actor, which often leads to unintuitive behaviors.. It is very difficult to debug an Akka program once it is implemented; on the contrary, if the design is bullet-proof, implementing it does not take too long.

Test Cases

Test cases will be syntactically similar to the ones as in Project 1. Both sequential and concurrent test cases should work in Phase 1 of Project 2 itself. (The to-be-omitted end-points are not to be used in the test cases.)

Code folder

Your submitted zip should contain a total of four folders. Two folders will contain the Account and Wallet services. These will be the same as your old (Project 1 Phase 2) implementations, except that you must remove any bugs that you are now aware of.

The third folder will contain your non-concurrent as well as concurrent test-cases. You can just put your old Phase 1 and Phase 2 test cases here (modulo that these should not use the

omitted end-points, and should achieve the same result in other ways). It is optional to add new test cases (unless a penalty was put on you in Project 1 for insufficient test cases). You should not include in this folder the public and private test cases shared by us.

The fourth folder will contain your Akka project. As with the other projects, this project should contain a Dockerfile at the project's root folder. You are required to write the Dockerfile using any linux-based base image (such as `openjdk:17`, `openjdk:21` or `ubuntu`). The machine we will use to run your container is linux-based, and your Docker image needs to be compatible with this. Do not assume that any dependencies or applications are available on our machine. Your Dockerfile should install into the container (using RUN directives) JDK, Maven and any other programs or libraries or dependencies that are needed to build and run your application. Your project checked-in in your repo should not contain any jar file or .class files, as those should be generated by the Dockerfile when we build your image.

As before, we will run the following commands to start your Marketplace container:

Build the docker image
```
docker build -t Marketplace-service .
```

Build the docker image
```
docker run -p 8081:8080 --rm --name marketplace
--add-host=host.docker.internal:host-gateway marketplace-service
```

Stop the container
```
docker stop marketplace
```

Remove the docker image
```
docker image rm marketplace-service
```

We will use the same commands as in Project 1 Phase 1 to start/stop your other two containers.

After this, the test cases should be able to run.

Phase 1 Logistics

- The project will be in two phases. The description given above is for Phase 1 alone. We will add additional requirements later for Phase 2.
- Deadline for Project 2 Phase 1 will be **April 2nd 11.59 pm**.

Other alternatives to using HttpServer

One disadvantage of the HttpServer approach recommended above is that it is not very clean and elegant to extract information from a HTTP request, or to construct a HTTP response. HttpServer is somewhat of a low-level approach for this purpose.

An alternative approach is to use a blend of Spring and Akka. You could spawn the root actor and the Gateway actor before the first request is received (using a Service, or CommandLineRunner, or a Component – please figure this out). Each request is to be processed by a controller method, as in the Project 1. Each controller method should ask() the Gateway actor to achieve the actual work (as is done by the handle() method in the recommended approach mentioned above).

Another alternative method, which does not require a merger of Spring and Akka, is to use Akka Http. This is a native facility in Akka to start a http server, and interface between the Http server and the actors. In order to get started with Akka HTTP, you may want to refer to the Akka HTTP Quickstart Project. Download and run the project, learn the core features of Akkk HTTP from the project's code and from the document, and then perhaps extend/modify this project to implement your Markeplace project. Complete Akka HTTP documentation is available here.

The function startHttpServer (called by main() in QuickstartApp.java) is notable. It starts a HTTP server by calling the function newServerAt(). This HTTP server implicitly spawns a new thread whenever a request is received. And this thread uses the "route" returned by function `userRoutes()` in order to fulfil the request and return the response.

A note on the function `userRoutes()`. Your version of this program can be renamed as `marketplaceRoutes`. This function returns a route. A "route" in Akka HTTP is a specification of how to respond to HTTP requests. It is hierarchical in nature, as it nests request-path suffixes within request-path prefixes. In the quickstart example, the `userRoutes` function calls the function getUsers() when it receives a GET request for path /users, calls the function createUser() when it receives a POST request for path /users whose payload contains the JSON of a User, calls the function getUser() whenever it receives a GET request for a path of the form /users/<*user-name*>, and calls the function deleteUser() whenever it receives a DELETE request for a path of the form /users/<*user-name*>. The userRegistryActor is to be replaced with the Gateway actor in our setting.

Whichever option you go with, the Docker command we will use to start/stop your Marketplace service will remain the same as mentioned earlier in this doc.