Food Delivery Project Using Akka

In this project you will reimplement only the Delivery microservice in Akka. For the other two microservices (i.e., Restaurant and Wallet), you will simply reuse your previous Docker+Spring based microservices. Restaurant 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).

Background on Akka HTTP

You will need to use Akka HTTP to make your Akka program process HTTP requests. We did not cover Akka HTTP in class, so you will need to read up the required parts of the documentation on your own. You can also experiment with this Akka HTTP Quickstart program to learn to use Akka HTTP. Raghavan Komondoor created this program basically by downloading the Akka HTTP Quickstart program from the Akka documentation site and modifying just a single line: Line 91 in UserRoutes.java. The README file at the root folder (this file was also created by Raghavan) explains to you how you can use this program. Of course you may need to use more features of Akka HTTP in your project (especially in the "route creation" function) than what is used in this quickstart program. You may want to start implementing your project by creating a copy of this quickstart project, and then adding your code and deleting unnecessary code. This way you inherit the pom.xml file and the basic folder structure.

A note on the function `userRoutes` in the quickstart program: Your version of this program can be renamed as deliveryRoutes. 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. You can read more about routes in the documentation mentioned above, and also on many discussion forums online.

A note about `AskPattern.ask:` This is an alternative to .tell(), and is required to be used whenever the code sending a message is not an actor (meaning, it does not have an incoming message queue and cannot receive a response message back in a handler). For instance, in the quickstart program, the userRoutes function, which is sending the message to the userRegistryActor, is not an actor. AskPattern.ask implicitly creates an actor to receive the response (pointed to by "ref" in methods such as getUser in UserRoutes.java). Typically, we provide to AskPattern.ask a function that puts this actor into the "replyTo" field of a message, and AskPattern.ask then sends this message to the actor given as its first argument (userRegistryActor in the example). userRegistryActor can directly send a response message to the "replyTo" actor in the message it received, or it can pass this replyTo ActorRef to some other actor that can respond. Now, AskPattern.ask does not wait

for this response message. Instead it immediately returns a “future” (of type CompletionStage). In the route, we block on this future (using `onSuccess`), and when the response is received the future returns the received message for further processing. You can read more about AskPattern.ask in numerous online discussion forums.

The Delivery microservice

This will be a single-node Akka program (i.e., no cluster). It will listen at http://localhost:8081 to http requests from outside. The request and response format should be the same as in Project 1.

Required actors in your program

Similar to the userRegistryActor in the quickstart project, you should implement a `Delivery` actor. Just spawn one instance of this actor upfront before receiving any http requests.

You will need an `Agent` actor. You should spawn as many instances of this actor as there are agents in the initialData.txt file. These actors should also be spawned before the first http request is received. You could keep ActorRefs to these actors in a map, indexed by agent IDs, and the map could be part of the state of the Delivery actor. Let us call this map `agentRefs`. Each Agent actor should maintain its own internal state, such as the current status of the agent, which order the agent is now serving (if any), etc. The Delivery actor should **not** store **live** information about the statuses of agents in its internal state. We make this design decision since `FulfillOrder` actors (which we introduce below) will be independently contacting the agent actors.

You will need a `FulfillOrder` actor. Each instance of a FulfillOrder actor stores the current status of an order, as well as the agent (actor) who is delivering the order (once an agent has been assigned to the order). The Delivery actor should maintain a map of FulfillOrder actors in its internal state, indexed by order IDs.

Suggested actions in your program

We provide a high-level design here. The detailed design of the exact set of messages to use, and the exact flow of sending and receiving messages, is up to you.

Every incoming http request could result in an appropriate message being sent to the Delivery actor. Hence, the Delivery actor will be the primary contact point from the HTTP routes function.

For an incoming “RequestOrder” message from the client, the Delivery actor should generate a fresh order ID, spawn a FulfillOrder actor, and rightaway return the order ID to the client. The Delivery actor should not perform the time-consuming task of finding an agent to deliver the order, as that would prevent the Delivery actor from processing subsequent HTTP requests in quick succession. For each freshly spawned FulfillOrder actor, the Delivery actor should remember somehow that it is waiting for an agent.

The logic of the FulfillOrder actor should be basically the same as specified in Project 1 under the /requestOrder end-point. One change is that the order ID is to be generated by the Delivery actor itself before spawning the FulfillOrder actor. The newly spawned FulfillOrder actor first sets its order status to *unassigned*. It then performs the remaining steps like asking the Wallet microservice to deduct the amount, and asking the Restaurant microservice to accept the order, etc. For this, it can use the usual Java libraries to make http requests. If the order cannot be fulfilled due to shortage of inventory in the restaurant or due to insufficient wallet balance, the FulfillOrder actor should set its status to *rejected* (this is a new status, not used so far in Project 1 or in Project 2). It should then stay in this status forever.

If the Restaurant microservice accepts the order, the FulfillOrder actor should contact the agent actors to see if any one is available to deliver the order. If none are available, it should wait to be notified of the availability of an agent. (When the FulfillOrder actor is spawned, the DeliveryActor can pass to it via the constructor a pointer to the agentRefs map. Passing a pointer to internal state is ok in this setting because this map is read-only after it is initialized.) Whenever a waiting FulfillOrder actor is notified of an available agent actor, it should go ahead and contact the agent actor to see if it is available to deliver the order.

Whenever a FulfillOrder actor finally assigns the order to an agent, it can (a) set its order status to *assigned*, and (b) notify the Delivery actor about this, so that the Delivery actor knows that this FulfillOrder actor is no longer waiting for an agent. ~~Note that if an order cannot be fulfilled due to rejection by the Restaurant microservice or Wallet microservice, then this order’s status should remain *unassigned* forever. This is a difference from Project 1, as in Project 1 the order-id itself is created only if Restaurant and Wallet approve of the order.~~

Whenever an agent actor’s status changes to *available*, it could send a message to the Delivery actor to say that it is now available. The Delivery actor could in turn notify one of the waiting FulfillOrder actors about this agent.

The Delivery actor could forward incoming AgentSignIn and AgentSignOut messages from the client to the corresponding agent actor. The Delivery actor could itself provide the response to the client.

The Delivery actor could forward an incoming OrderDelivered message from the client to the corresponding FulfillOrder actor and respond to the client. The FulfillOrder actor could update its internal order status to *delivered*, and could notify the agent who delivered the order that they are now *available.*

When the Delivery actor receives incoming messages from the client corresponding to the “GET /order/*num*” or “GET /agent/*num*” end points, it would need to forward these messages to the corresponding FulfillOrder or Agent actor, and let them respond to the client.

When the Delivery actor receives the ReInitialize message from the client, it can get rid of all FulfillOrder actors, and tell each Agent actor to sign out.

You can assume for simplicity that no messages will be lost in the system.

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, which often leads to unintuitive differences in the order in which messages are actually received compared to what one would first expect (we have discussed some of these scenarios in class). 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.

When you start implementation, since the http part is new, first implement a skeleton version of your code that does not have most of the logic, but communicates correctly with the client and with the other two microservices. Test this skeleton version using appropriate test cases. This way you will identify early any issues with http communication, rather than detect such issues on the last day.

Test Cases

Test cases will be syntactically similar to the ones as in Project 1. One key difference is that an order ID gets generated and returned for a requestOrder even if Restaurant or Wallet do not approve of the order, and information about such an order ID can be requested from the test case later. Both sequential and parallel test cases should work. Due to the asynchronous nature of the Akka implementation, the assertions/conditions checked in the test cases may need to be different in the Akka test cases than in the Spring test cases, as some of the assertions under Spring may no longer pass under asynchrony.

Logistics

- The project will be in two phases. The description given above is for the final deliverable (i.e., at the end of Phase 2). For Phase 1, you can do a simplified version of the same project, wherein there are no Agent actors. Instead, each FulfillOrder actor can mark its order status as *delivered* as soon as Restaurant and Wallet approve of the order. The /agentSignIn, /agentSignOut, /agent/num, and /orderDelivered end-points need not be supported. In the response to the /order/num end-point, agentId need not be present.
- In each phase you should upload a zip of a folder that contains the three projects as subfolders, and a fourth subfolder containing your test cases.
    - The zip name should be of the format [SR.No_of_Member1]_[SR.No_of_Member2] as submitted while forming the team. For Eg. If the team is Team Member1 (12345) and Team Member2 (67890) then zip name should be 12345_67890.zip. If only one team member is there For e.g. Team Member1 (12345), then zip name should be 12345_12345.zip
    - The four subfolder names should be "**Restaurant**", "**Delivery**", "**Wallet**", "**Tests**" . Please follow this common nomenclature since we will be doing automated checking of your project.
    - If you have used some additional packages (hopefully none) then mention them in a Readme file, and also include instructions to install these packages.
    - Please write short description about each private testcase at top of testcase file. So it will be easy for us to understand your testcases if we need to check your private testcases.
- No need for you to provide any launch script or teardown script. Rather, we will be using the commands mentioned in this file to compile and run your project: https://indianinstituteofscience-my.sharepoint.com/:t:/g/personal/raghavan_iisc_ac_in/Ecpv3BxKXQ5LrLWxw-bFwgQBh0YeX04zDndLHjoHDJwRkw?e=Uwbyex
  So, please make sure everything works using these commands.
- The initialData.txt file will be available at the root location in all three project subfolders. docker should mount this file using the “-v” option, while mvn should give the location of this file to the Akka program as a command-line argument using `-Dexec.args=”./initialData.txt”`.
- Grading criteria will be similar to Project 1. Each of the two phases will carry equal marks.
- Deadline for Phase 1 will be **March 31st 11.59 pm**, while for Phase 2 it will be **Apr. ~~17th~~ 19th 11.59 pm**. This time there will not be any extensions to these dates under any circumstances, so please definitely keep this in mind.