# PoDS 2021, 1st Course Project: Cab Hailing Application using Spring

The objective of this project is to develop a taxi (cab) hailing application (similar to Ola, Uber, etc.) The application will be organized as a set of RESTful services. In other words, there is no need for a UI (you can develop a UI also additionally if you are interested). You should make the following simplifying assumptions for the sake of this project:

- All locations are on the x-axis. That is, each location can be described by a single (non-negative) integer.
- There is a fixed set of customers and cabs, specified initially in a text file when the application starts.
- Each customer maintains a wallet with the cab company, and all rides are paid from the wallet only.
- The payment for any trip is deducted from the customer's wallet before the ride starts.
- Cabs can sign-in and sign-out as per the wish of the drivers. Only a signed-in cab can offer rides.
- A cab stays at the same location after dropping off a customer until it is called somewhere else to start a new ride, or until it signs out for the day.

## Required services and their end-points

1. Cab: *// represents the cabs*

    *The "major" states that a cab can be in at any given time are* signed-out *or* signed-in*. Within the* signed-in *state, there are further "minor" states, namely,* available, committed, *and* giving-ride. *When a cab signs in for the day, it enters the* available *state. The rest of the state transitions are explained below.*

    boolean requestRide(int cabId, int rideId, int sourceLoc, int destinationLoc)

        *RideService.requestRide invokes this request, to request a ride on cabId to go from "sourceLoc" to "destinationLoc". rideId is a unique ID that identifies this request. The response to this request should be* true *iff the cab is accepting the request. The cab should accept the request iff cabId is a valid ID and cabId is currently in* available *state and is* interested *in accepting the request. Once a* true *response is sent, the*

*cabId enters the* committed *state, whereas if a* false *response is sent it remains in* available *state.*

*In your implementation you should answer the "is interested?" question by making each cab accept the first request that it receives after it signs in on any day, and then, considering all requests it receives while it is in* available *state, it should be disinterested in every alternate request among these requests.*

*This end-point needs to be* isolated. *That is, while one requestRide request is being handled for a cabId, if another requestRide request comes in for the same cabId it should be made to wait until the first request is responded to. Note, this matters only for Phase 2 of the project, not for Phase 1.*

boolean rideStarted(int cabId, int rideId)

*This request is triggered by RideService.requestRide. If cabId is valid and if this cab is currently in* committed *state due to a previously received Cab.requestRide request for the same rideId, then move into* giving-ride *state and return* true, *otherwise do not change state and return* false*.*

boolean rideCanceled(int cabId, int rideId)

*This request is triggered by RideService.requestRide. If cabId is valid and if this cab is currently in* committed *state due to a previously received Cab.requestRide request for the same rideId, then enter* available *state and return* true, *otherwise do not change state and return* false*.*

boolean rideEnded(int cabId, int rideId)

*This request is triggered by the driver when the ongoing ride ends. If cabId is valid and if this cab is currently in* giving-ride *state due to a previously received Cab.rideStarted request for the same rideId, then enter* available *state and send request RideService.rideEnded and return* true, *otherwise do not change state and return* false*. A ride is always assumed to end at the originally specified destination of the ride.*

boolean signIn(int cabId, int initialPos)

*Cab driver will send this request, to indicate his/her desire to sign-in with starting location initialPos. If cabId is a valid ID and the cab is currently in* signed-out *state, then send a request to RideService.cabSignsIn, forward the response from RideService.cabSignsIn back to the driver, and transition to* signed-in *state iff the response is* true. *Otherwise, else respond with "false" and do not change state.*

boolean signOut(int cabId)

*Cab driver will send this request, to indicate his/her desire to sign-out. If cabId is a valid ID and the cab is currently in* signed-in *state, then send a request to RideService.cabSignsOut, forward the response from RideService.cabSignsOut back to the driver, and transition to* signed-out *state iff the response is* true. *Otherwise, else respond with -1 and do not change state.*

int numRides(int cabId)

*To be used mainly for testing purposes. If cabId is invalid, return -1. Otherwise, if cabId is currently* signed-in *then return number of rides given so far after the last sign-in (including ongoing ride if currently in* giving-ride *state), else return 0.*

2. RideService: *// This service represents the cab-hailing company.*
*// The service should internally keep its own record of the current*
*// states and positions of the cabs.*

boolean rideEnded(int rideId)

*Cab uses this request, to signal that rideId has ended (at the chosen destination). Return true iff rideId corresponds to an ongoing ride.*

boolean cabSignsIn(int cabId, int initialPos)

*Cab cabId invokes this to sign-in and notify the company that it wants to start its working day at location "initialPos". Response is true from the company iff the cabId is a valid one and the cab is not already signed in.*

boolean cabSignsOut(int cabId)

*Cab uses this to sign out for the day. Response is true (i.e., the sign-out is accepted) iff cabId is valid and the cab is in* available *state.*

int  requestRide(int custId, int sourceLoc, destinationLoc)

*Customer uses this to request a ride from the service. The cab service should first generate  a globally unique rideId corresponding to the received request. It should then try to find a cab (using Cab.requestRide) that is willing to accept this ride. It should request cabs that are currently in* available *state one by one in increasing order of current distance of the cab from sourceLoc.  The first time a cab accepts the request, the service should calculate the fare (the formula for this is described later) and attempt to deduct the fare from custId's wallet. If the deduction was a success, send request Cab.rideStarted to the accepting cabId and then respond to the customer with the generated rideId, else send request Cab.rideCanceled to the accepting cabId and then respond with -1 to the customer. If three cabs have been requested and all of them reject, then respond with -1 to the customer. If fewer than three cabs have been contacted and they all reject the requests and there are no more cabs available to request that are currently signed-in and not currently giving a ride, respond with -1 to the customer.*

*Whenever responding with a valid rideId to the customer (i.e., not -1), also include the cabId and fare in the response. These three items should be separated by spaces. For e.g., "5561 101 1000" is a response. Here, 5561 is the rideId, 101 is the cabId, and 1000 is the fare.*

*The fare for a ride is equal to the distance from the accepting cab's current location to sourceLoc plus the distance from sourceLoc to destinationLoc, times 10 (i.e., Rs. 10 per unit distance).*

String getCabStatus(int cabId)

*This end-point is mainly to enable testing. Returns a tuple of strings indicating the current state of the cab* (signed-out/available/committed/giving-ride)*, its last known position, and if currently in a ride then the custId and destination location. The elements of the tuple should be separated by single spaces, and the tuple should not have any beginning and ending demarcators. Last known position is the source position of the current ride if the cab is in* giving-ride *state, is the sign-in location if it has signed in but never entered giving-ride state after signing in, is the destination of the last ride if it is currently in* available *or* committed *state and gave a ride after sign-in, and is -1 if it is in signed-out state.*

void reset()

*This end-point will be mainly useful during testing. This end-point should send Cab.rideEnded requests to all cabs that are currently in* giving-ride *state, then send Cab.signOut requests to all cabs that are currently in sign-in state.*

3. Wallet:

int getBalance(int custId)

*returns current wallet balance of custId*

bool deductAmount(int custId, int amount)

*If custId has balance >= amount, then reduce their balance by "amount" and return true, else return false. This service is used by RideService.requestRide.*

bool addAmount(custId, int amount)

*Inverse of deductAmount.*

*Both deductAmount and addAmount need to be processed* in an isolated manner *for a custId (i.e., different requests for the same custId should not overlap in time). Again, this isolation requirement matters only in Phase 2.*

void reset()

*Reset balances of all customers to the "initial" balance as given in the input text file (more details about this text file below). This end-point is mainly to help enable testing.*

## Requirements from your implementation

- Each of the three services described above must be a separate microservice, i.e., must be packaged and deployed as a separate container.
- Use Spring to implement each microservice.
- For simplicity, we will use only HTTP GET requests (for all end points). The URLs for the Cab, RideService, and Wallet microservices should be, respectively, localhost:8080, localhost:8081, and localhost:8082 (use the -p option of "docker run" to achieve this). For each end-point, the path after the

"/" in the URL should be the name of the end-point, and the request parameters should match the parameters mentioned above (in name and in order). For instance, the GET request "http://localhost:8082/addAmount?custId=110&amount=1000" corresponds to telling Wallet.addBalance to add 1000 rupees to the wallet of custId 110. Note, for simplicity, we are not following many of the recommendations for RESTful services that we studied in class!

## Test cases

[*Note:* This Section pertains to Phase 1 of the project only. ]

You will be needing test cases to test your implementation. Each test case will be a "sh" shell script, and each script will make a sequence of requests with specific request-parameter values (using the program "curl", as demonstrated in the various Spring demos referred to in the class slides). Test cases will typically raise requests to all the end-points mentioned above *except* Cab.requestRide, Cab.rideStarted, Cab.rideCancelled, RideService.rideEnded, RideService.cabSignsIn, RideService.cabSignsOut, Wallet.deductAmount, and Wallet.addAmount (as these services are to be invoked internally from other services). Each test case should check the response to every request and should finally print a "Passed" message iff the response to every request is as expected.

We will be providing you soon a small set of sample test cases (which we call "public test cases"). These will only be samples -- you are expected to make your own exhaustive set of test cases. Normally each test case should test a specific interesting scenario. A single test case should not test multiple independent scenarios, as you should prefer to have separate test cases for separate scenarios. Each test case should assume when it starts that no rides are ongoing, all cabs are signed out, and the wallet balance of every customer is the "initial" balance as specified in the input text file; correspondingly, every test case should invoke RideService.reset and Wallet.reset at the end, so that the next text case to execute gets to start from a clean slate. **You should aim to create a good set of test cases that test numerous interesting scenarios, corner cases, etc.**

**Phase 1**: Implement and deploy the three containers corresponding to the three services as described above. You don't need to use a database in this phase -- each microservice can keep its data in in-memory data structures. You can use plain http in this phase for all the requests.

The set of valid cabIds, valid custIds, and initial wallet balance of each customer to be read from an input text file on the host that will be made available to all the

containers with the “--mount” or “-v” option of “docker run”. This file’s name should be `IDs.txt`, and its path on both the host and within the container is up to you. The containers should treat this file as read-only. When the containers start running they should initialize all wallet balances to the initial balance given in the text file, and should treat all cabIds as if they are in signed-out state. The format of the input file is given in Appendix A.

**Phase 2**: This phase needs the following enhancements on top of Phase 1’s work.

- Deploy the containers using kubernetes (minikube).
- Have a variable number of RideService instances, from 1-4, which changes at runtime based on the load. The RideService instances should be stateless, so that any request can be directed to any instance (kubernetes should manage the forwarding of requests to RideService instances using load balancing). All RideService instances should use a common database service which is deployed separately outside. The RideService instances should keep all information that was in-memory in Phase 1 (e.g., information about ongoing rides, information about the current positions of cabs, etc.), in this database. You can use any kind of database system as you like, and should use Spring Data JPA to access the database. The database needs to support concurrent queries properly (this point is discussed more later).
- For simplicity, have a single instance each of the Cab and Wallet microservices. Also, for simplicity, these microservices can keep all their data in in-memory data structures.
- Recall, some of the requests need to be processed in an *isolated* manner. This has been indicated in the description in earlier pages. It was not important for Phase 1, but it is important in Phase 2. You can use Java synchronized blocks or any other suitable mechanism to impose isolation.
- You will need to support concurrently running test cases (i.e., multiple test scripts running concurrently from different terminal windows). This means multiple requests could be received by and could be processed by Spring controllers concurrently. Therefore, you need to ensure that all shared data structures in the Cab and Wallet microservice are used in a thread-safe manner. We have imposed the *isolation* requirement for this reason only, but you should think more about whether this isolation suffices or whether you need to take additional precautions in the code. Regarding the RideService microservice, you should check and ensure that the database provides ACID (or at least, the ACI properties) in the presence of concurrent requests.
- See the section titled “A template for a concurrent test script for Phase 2” in Appendix B to see how to construct a concurrent test script.

Resources on Spring Data JPA:

- Sample code used : https://github.com/geetam/pods_springdatajpa_demo
- A quick tutorial to get started: https://spring.io/guides/gs/accessing-data-jpa/
- A more detailed tutorial: https://www.petrikainulainen.net/spring-data-jpa-tutorial/

## Appendix A: Input text file format

Sample input file:

```
****
101
102
103
104
****
201
202
203
****
10000
```

The first section in the file contains the cabIDs, the second section contains the custIDs, while the last section contains the initial wallet balance of all customers. The four *'s are the section-begin markers. Number of digits in each ID can be between 1 to 10 digits.

## Appendix B: Public test cases

All public test cases in this document are wrt the sample input file given in Appendix A.

### Test case Pb1

```
#! /bin/sh
# this test case checks whether a customer's request
# gets rejected if only one cab has signed in but it is busy.

# reset RideService and Wallet.
# every test case should begin with these two steps
```

```
curl -s http://localhost:8081/reset
curl -s http://localhost:8082/reset

testPassed="yes"

#cab 101 signs in
resp=$(curl -s “http://localhost:8080/signIn?cabId=101&initialPos=0”)
if [ "$resp" = "true" ];
then
      echo "Cab 101 signed in"
else
      echo "Cab 101 could not sign in"
      testPassed="no"
fi

#customer 201 requests a ride
rideId=$(curl -s \
“http://localhost:8081/requestRide?custId=201&sourceLoc=2&destinationLoc=10”)
if [ "$rideId" != "-1" ];
then
      echo "Ride by customer 201 started"
else
      echo "Ride to customer 201 denied"
      testPassed="no"
fi

#customer 202 requests a ride
rideId=$(curl -s \
“http://localhost:8081/requestRide?custId=202&sourceLoc=1&destinationLoc=11”)
if [ "$rideId" != "-1" ];
then
      echo "Ride by customer 202 started"
      testPassed="no"
else
      echo "Ride to customer 202 denied"
fi

echo "Test Passing Status: " $testPassed
```

Note, the final “echo” above is important, and should follow the exact same format as shown above in all your test cases. The final  echo is expected to print “yes” for all test cases. The other intermediate echo’s are up to you.

## Test Case Pb2

```
#! /bin/sh
# this test case to track cab status

# every test case should begin with these two steps
curl -s http://localhost:8081/reset
curl -s http://localhost:8082/reset
```

```
testPassed="yes"

#Step 1 : Status of a signed-out cab
resp=$(curl -s \
“http://localhost:8081/getCabStatus?cabId=101”)
if [ "$resp" != "signed-out -1" ];
then
    echo "Invalid Status for the cab 101"
    testPassed="no"
else
    echo "Correct Status for the cab 101"
fi

#Step 2 : cab 101 signs in
resp=$(curl -s “http://localhost:8080/signIn?cabId=101&initialPos=100”)
if [ "$resp" = "true" ];
then
    echo "Cab 101 signed in"
else
    echo "Cab 101 could not sign in"
    testPassed="no"
fi

#Step 3 : Status of a signed-in cab
resp=$(curl -s \
“http://localhost:8081/getCabStatus?cabId=101”)
if [ "$resp" != "available 100" ];
then
    echo "Invalid Status for the cab 101"
    testPassed="no"
else
    echo "Correct Status for the cab 101"
fi


#Step 4 : customer 201 requests a ride
rideId=$(curl -s \
“http://localhost:8081/requestRide?
custId=201&sourceLoc=110&destinationLoc=200”)
if [ "$rideId" != "-1" ];
then
    echo "Ride by customer 201 started"
else
    echo "Ride to customer 201 denied"
    testPassed="no"
fi

#Step 5 : Status of a cab on ride
resp=$(curl -s \
“http://localhost:8081/getCabStatus?cabId=101”)
if [ "$resp" != "giving-ride 110 201 200" ];
then
    echo "Invalid Status for the cab 101"
```

```
    testPassed="no"
else
    echo "Correct Status for the cab 101"
fi

#Step 6 : End ride1
resp=$(curl -s \
“http://localhost:8080/rideEnded?cabId=101&rideId=$rideId”)
if [ "$resp" = "true" ];
then
    echo $rideId1 " has ended"
else
    echo "Could not end" $rideId1
    testPassed="no"
fi


#Step 7 : #Status of a cab after a ride
resp=$(curl -s \
“http://localhost:8081/getCabStatus?cabId=101”)
if [ "$resp" != "available 200" ];
then
    echo "Invalid Status for the cab 101"
    testPassed="no"
else
    echo "Correct Status for the cab 101"
fi


echo "Test Passing Status: " $testPassed
```

## A template for a concurrent test script for Phase 2

Shell script `main`:

```
#! /bin/sh
# every test script should begin with these two steps
curl -s http://localhost:8081/reset
curl -s http://localhost:8082/reset

# Run two test cases in parallel
sh sh1 &
sh sh2
# sh1 creates the output file sh1out, which contains fares
# of all rides given in sh1. Similarly, sh2out.

wait

totalFare=0
for i in $(cat sh1out sh2out);
do
```

```
  totalFare=$(expr $totalFare + $i)
done
# totalFare contains the sum cost of all rides

# Now check if the current total balance
# in all wallets is equal to
# original total balance in all wallets (which is a constant)
# MINUS totalFare.
# Print “Test Passing Status: yes” if yes,
# else print “Test Passing Status: no”.
You need to fill the code for the check above
```

Shell script sh1:

```
rm -f sh1out

#Step 1 : cab 101 signs in
resp=$(curl -s “http://localhost:8080/signIn?cabId=101&initialPos=100”)
if [ "$resp" = "true" ];
then
    echo "Cab 101 signed in"
else
    echo "Cab 101 could not sign in"
fi

# Step 2: customer 201 requests a cab
rideDetails=$(curl -s \
“http://localhost:8081/requestRide?
custId=201&sourceLoc=110&destinationLoc=200”)
rideId=$(echo $rideDetails | cut -d' ' -f 1)
cabId=$(echo $rideDetails | cut -d' ' -f 2)
fare=$(echo $rideDetails | cut -d' ' -f 3)
if [ "$rideId" != "-1" ];
then
    echo "Ride by customer 201 started"
    echo $fare >> sh1out
else
    echo "Ride to customer 201 denied"
fi
```

Shell script sh2:

```
rm -f sh2out

#Step 1 : cab 101 signs in
resp=$(curl -s “http://localhost:8080/signIn?cabId=101&initialPos=100”)
if [ "$resp" = "true" ];
then
    echo "Cab 101 signed in"
else
    echo "Cab 101 could not sign in"
```

```
fi

# Step 2: customer 202 requests a cab
rideDetails=$(curl -s \
“http://localhost:8081/requestRide?
custId=202&sourceLoc=90&destinationLoc=200”)
rideId=$(echo $rideDetails | cut -d' ' -f 1)
cabId=$(echo $rideDetails | cut -d' ' -f 2)
fare=$(echo $rideDetails | cut -d' ' -f 3)
if [ "$rideId" != "-1" ];
then
    echo "Ride by customer 202 started"
    echo $fare >> sh2out
else
    echo "Ride to customer 202 denied"
fi
```

Notes about the test script above:

- You will run the `main` test script. `main` in turn invokes `sh1` and `sh2`.
- In the template above we run two test scripts in parallel. However, in the private test scripts, we may run any number of test scripts in parallel. To run three scripts, e.g., you can use: `sh sh1 & sh sh2 & sh sh3`
- Note, we no longer test for the passing or failing of each step in each script. Due to non-deterministic interaction between the parallel scripts, it is hard to predict which steps will pass and which steps will fail.
- However, overall the system should preserve some aggregate properties. One such aggregate property is that the total amount of money reduced from all wallets should be equal to the total fare of all rides given. The test script given above tests this aggregate property. In the private test cases we may test any other aggregate property that ought to hold across all possible runs.
- In the private test cases, we may use a loop in each of the files `sh1`, `sh2`, etc., in order to send a large number of requests back to back. We will then ask you to run “ps aux” to see if all the containers (all replicas of RideService) are using significant CPU time.
- Your Phase 2 implementation must continue to satisfy the original sequential test cases (after they are updated to account for the new response format for RideService.requestRide).