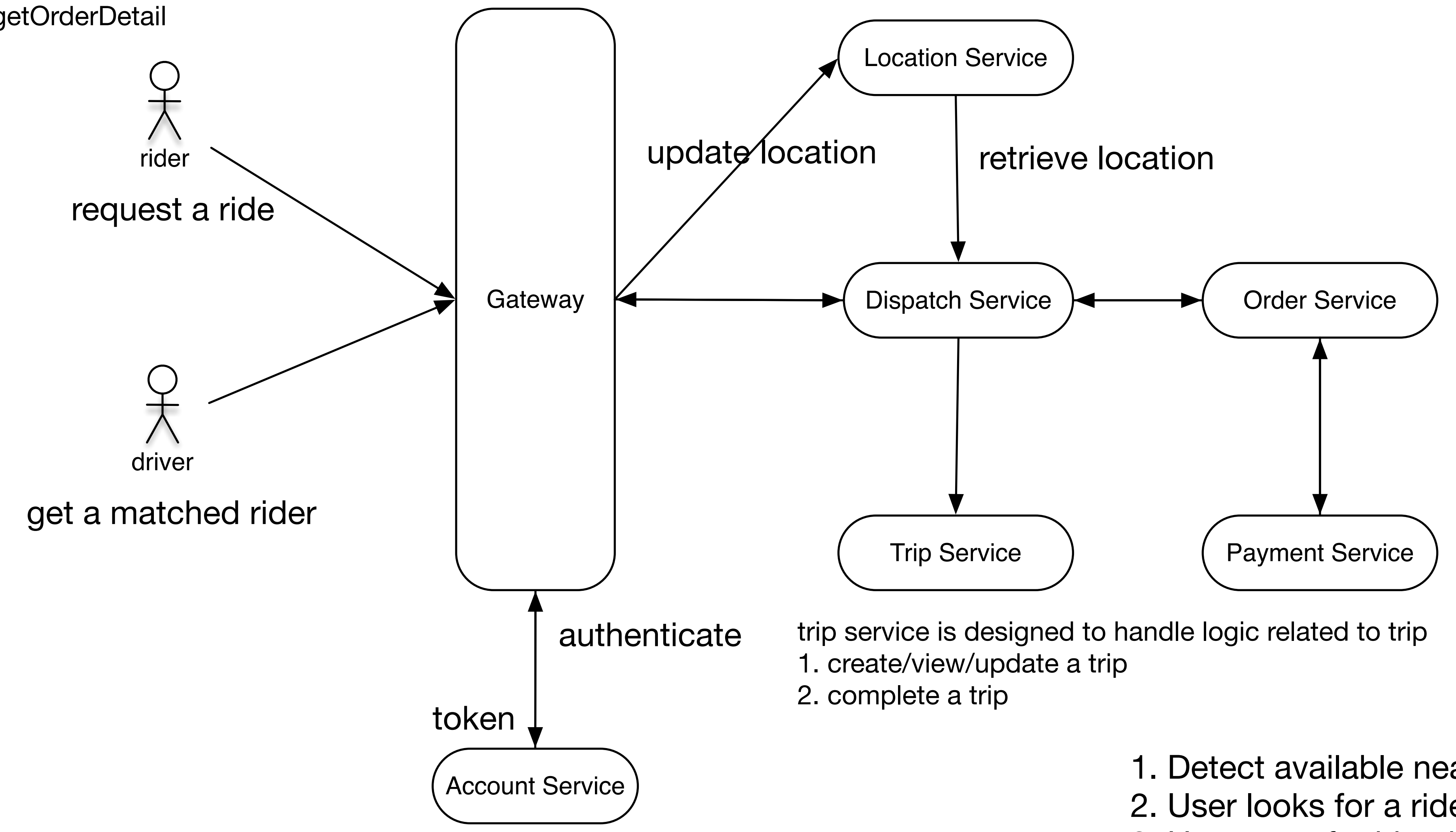
Uber

dispatch service is designed to handle logic related to ride request
1. request a ride - called by a rider
2. check a ride - called by a rider periodically to check if a driver has been found
3. complete a ride - called by a driver to complete the trip
4. get ride info - called by driver
5. update ride - called by driver to accept or decline a trip
6. createOrder
7. notify driver and rider when an order is successful charged
8. getOrderDetail

Geospatial 算法:
http://blog.notdot.net/2009/11/Damn-Cool-Algorithms-Spatial-indexing-with-Quadtrees-and-Hilbert-Curves
https://marcin-chwedczuk.github.io/iterative-algorithm-for-drawing-hilbert-curve
http://www.diva-portal.org/smash/get/diva2:1027550/FULLTEXT02 Chapter 5.10, page 37
http://itsumomono.blogspot.com/2015/07/poi.html

location service is designed to handle logic related to drivers and driver locations.
1. create/view a driver
2. update driver location - POST /drivers/{id}/location
3. view all historical locations - GET /drivers/{id}/locations
4. view current location - GET /drivers/{id}/location/current



order service is called when a trip is completed
1. create/update/get an order

trip service is designed to handle logic related to trip
1. create/view/update a trip
2. complete a trip

account management
authentication
authorization

DB schema:

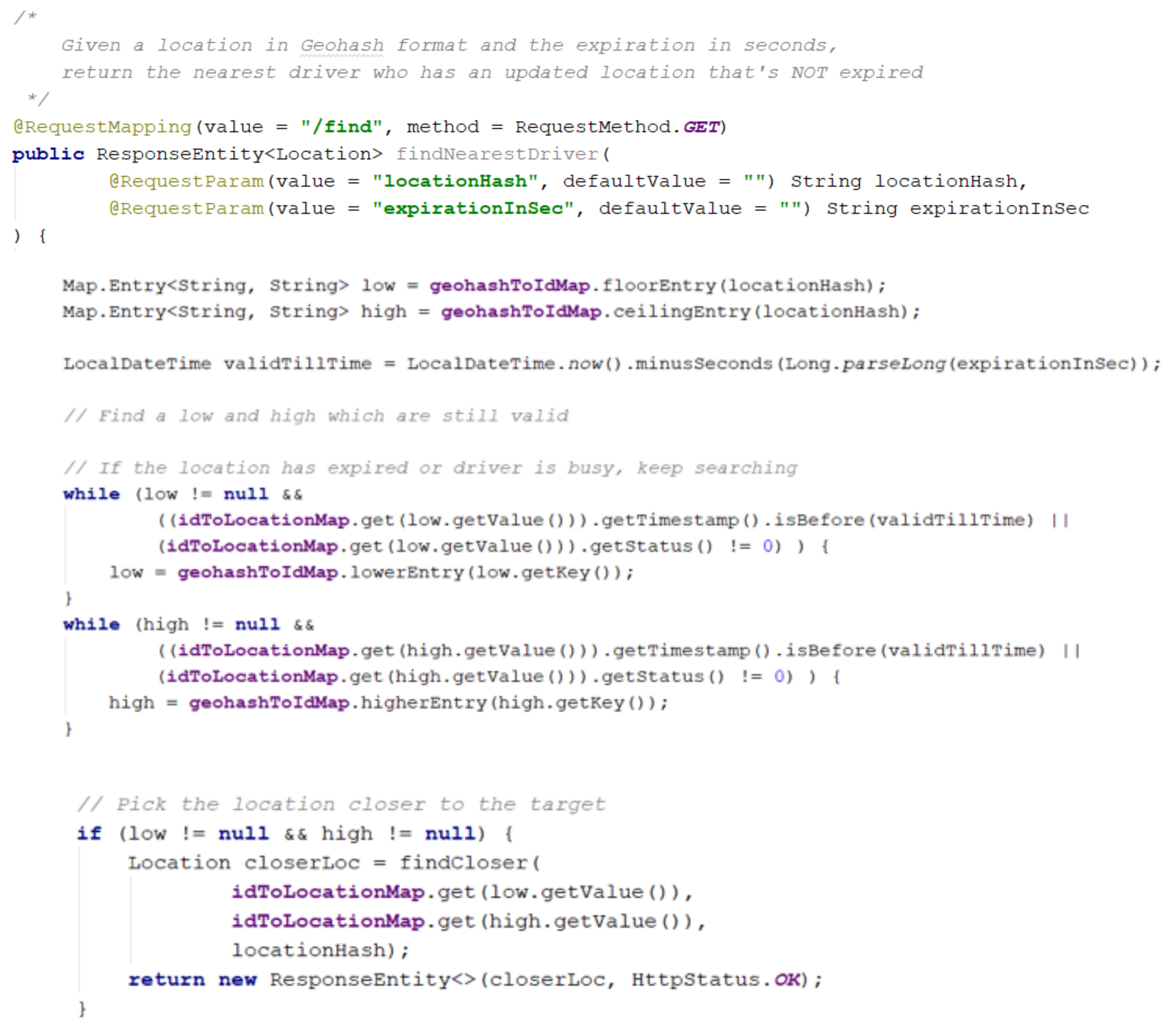
Drivers: id, firstName, lastName, address, phone, createdOn, **isActive**
Riders: id, firstName, lastName, address, phone, **payment**, createdOn
Trips: id, driver_id, rider_id, origin, destination, **status (**driver status: available, pending acceptance, accepted**)**, createdOn
Driver_Location:
- SQL: driver_id, latitude, longitude, updatedOn
- NoSQL (redis, key-value store):
  - key: driver_id
  - value: {latitude, longitude, updatedOn, status, trip_id}

Orders: order_id, trip_id, price, payment, status, createdOn, updatedOn

1. Detect available nearby vehicles
2. User looks for a ride
3. User pays for his trip
4. Calculate time (time = distance / speed, might be provided by Google Map APIs)

main scenarios:
driver: report locations to location service
1. report locations to location service
2. accept a dispatch request
3. complete a dispatch request

rider: ask dispatch service to find a driver
1. rider initiates a request
2. dispatch service finds a nearest driver
   * read from Redis?
3. driver accepts the request



```
/*
    Given a location in Geohash format and the expiration in seconds,
    return the nearest driver who has an updated location that's NOT expired
 */
@RequestMapping(value = "/find", method = RequestMethod.GET)
public ResponseEntity<Location> findNearestDriver(
        @RequestParam(value = "locationHash", defaultValue = "") String locationHash,
        @RequestParam(value = "expirationInSec", defaultValue = "") String expirationInSec
) {

    Map.Entry<String, String> low = geohashToIdMap.floorEntry(locationHash);
    Map.Entry<String, String> high = geohashToIdMap.ceilingEntry(locationHash);

    LocalDateTime validTillTime = LocalDateTime.now().minusSeconds(Long.parseLong(expirationInSec));

    // Find a low and high which are still valid

    // If the location has expired or driver is busy, keep searching
    while (low != null &&
            ((idToLocationMap.get(low.getValue())).getTimestamp().isBefore(validTillTime) ||
            (idToLocationMap.get(low.getValue())).getStatus() != 0) ) {
        low = geohashToIdMap.lowerEntry(low.getKey());
    }
    while (high != null &&
            ((idToLocationMap.get(high.getValue())).getTimestamp().isBefore(validTillTime) ||
            (idToLocationMap.get(high.getValue())).getStatus() != 0) ) {
        high = geohashToIdMap.higherEntry(high.getKey());
    }


     // Pick the location closer to the target
     if (low != null && high != null) {
        Location closerLoc = findCloser(
                idToLocationMap.get(low.getValue()),
                idToLocationMap.get(high.getValue()),
                locationHash);
        return new ResponseEntity<>(closerLoc, HttpStatus.OK);
     }
```