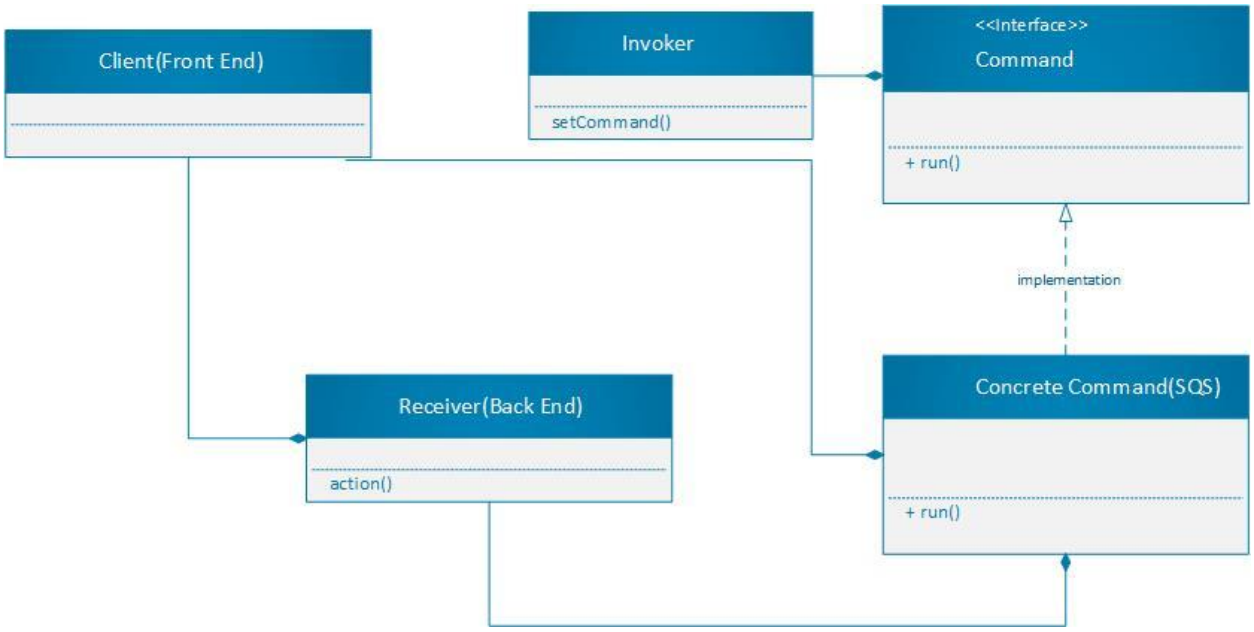
Book a Ride -- Careem

Below are the applications or actual functions of book a ride service.

- Customer will open car booking app.
- App can automatically identify the customer location based on the gps (using geo location api of android or core location api of ios) or customer can enter his location manually. The request can be send to server using node.js. Node.js framework very useful for maintaining persistent and reliable data base systems.
- The driver locations are updated to back end server asynchronously using push mechanism (pub-sub systems) with time frame.
- In real time the server will fetch nearest cabs based on customer location using shortest path algorithm or k-nearest neighbor's algorithm. And it will display early time arrival of the cab.

I have implemented above back end solution in java using both ***Command pattern*** and ***simple facade pattern***. Command pattern decouples an object making a request from the one that knows how to perform it. This is very good design solution for easy maintenance of multiple services in service oriented architecture. The facade pattern defines higher level interface to make subsystems easier to use.



The "SQS.java" file can act as Queue for storing actual events running on back end. Each event can be considered as a command.

For example, "BookCab.java", "CancelCab.java", "NearestCabs.java" are actual commands. "BookARide.java" can act as a simple facade for defining higher level functionality of the system. We can use multiple design solutions based on the requirements. Please check all the files in GitHub repository.

Code of SQS.java

```
import java.util.*;

import java.lang.*;

import java.io.*;


public class SQS {

	// We can create multiple queues of requests for different purposes. Here I can just be creating one for book a cab.

	Queue<Command> queue;


	public SQS() {

		queue = new LinkedList<Command>();

	}

	public void addCommand (Command command) {

		queue.add(command);

	}

	public void execute() {

		Iterator<Command> it = queue.iterator();

		while(it.hasNext()) {

			Command command = it.next();

			command.run();

		}

	}
```

}

Code of BookARide.java

```
import java.util.*;
import java.io.*;
import java.lang.*;


public class BookARide {


	GPS gps;
	Location location;
	User user;
	Command nearest;
	Command bookCab;
	Command payment;
	SQS sqs;


	public BookARide(GPS gps, Location location, User user, Command nearest, Command bookCab, Command payment, SQS sqs) {
		this.gps = gps;
		this.location = location;
		this.user = user;
		this.nearest = nearest;
		this.bookCab = bookCab;
		this.payment = payment;
		this.sqs = sqs;
	}
```

```
        public void bookARide() {
                sqs.addCommand(nearest);
                sqs.addCommand(bookCab);
                sqs.addCommand(payment);
                sqs.execute();
        }

}
```

For efficient access of user data and cab data from the data base we can directly store this information as ***documents representing JSON***. So that we can effectively retrieve the data using object id as a key. For this we can use either MongoDB or Couch DB.

We can also use replication to keep data geographically close to users. So that the data is always available and improve the read throughput. But for very large data sets or very high query throughput replication is not sufficient. We also need to break the data into partitions.

The main reason for wanting to partition data is scalability. Different partitions can be placed on different nodes in a shared-nothing cluster. Thus a large data set can be distributed across many disks, and the query load can be distributed across many processors. Partitioning is usually combined with replication, so that copies of each partition are stored on multiple nodes. This means that, even though each record belongs to exactly one partition, it may still be stored on several different nodes for fault tolerance.

Below are the challenges we have to address if the data is very high.

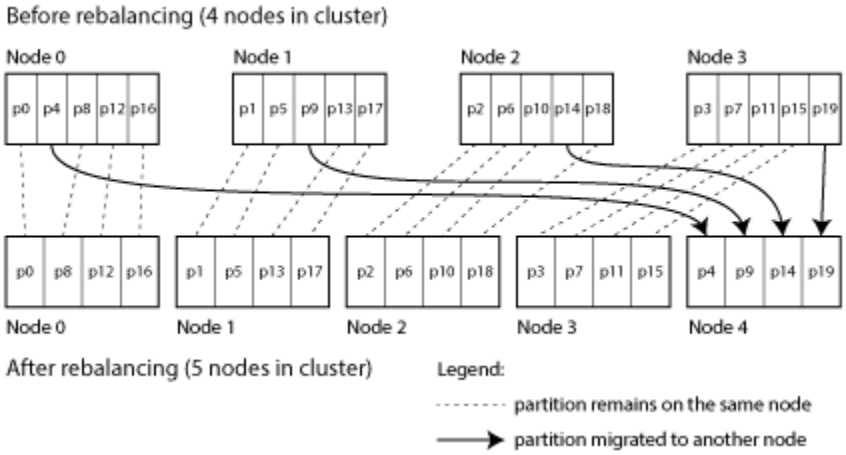
- ***How to address the issue of re-balancing between partitions?***
- ***How we can handle the service discovery(Request-routing) if data is distributed or partioned across multiple nodes?***

Below are the design solutions for the above scalability challenges.

1. No matter which partitioning scheme is used, rebalancing is usually expected to meet some minimum requirements.

- After re-balancing, the load (data storage, read and write requests) should be fairly shared between the nodes in the cluster.
- While re-balancing is happening the database should continue accepting reads and writes.
- We don't have to overload the network by moving more data.

We can address this issue by using multiple methods like fixed number of partitions of the data. Like we have to assign more load to powerful machines. Or we can partition dynamically based on the key range(HBase).



2. We have now partitioned our dataset across multiple nodes running on multiple machines. But there remains an open question: when a client wants to make a request, how does it know which node to connect to? As partitions are rebalanced, the assignment of partitions to nodes changes.
    - We can address this issue by allow clients to contact any node (e.g. via a round robin load balancer). If that node coincidentally owns the partition to which the request applies, it can handle the request directly. Otherwise it forwards the request to the appropriate node.
    - Or we can send all requests from clients to a routing tier first, which determines the node that should handle the request and forwards it accordingly. This routing tier does not itself handle any requests, it only acts as a partition aware load balancer.
    - Zookeeper is often used for service discovery, that is, to find out which IP address you need to connect to in order to reach a particular service. In cloud datacenter environments, where it is common for virtual machines to continually come and go, you often don’t know the IP addresses of your services ahead of time. Instead, you can configure your services such that when they start up, they register their network endpoint in a service registry, where they can then be found by other services.

We can also use a separate data access layer (persistent layer) to handle all the updates related to data base. So that the business or application logic does not depend on any particular data base.

For analytics we can go with batch processing using map reduce.