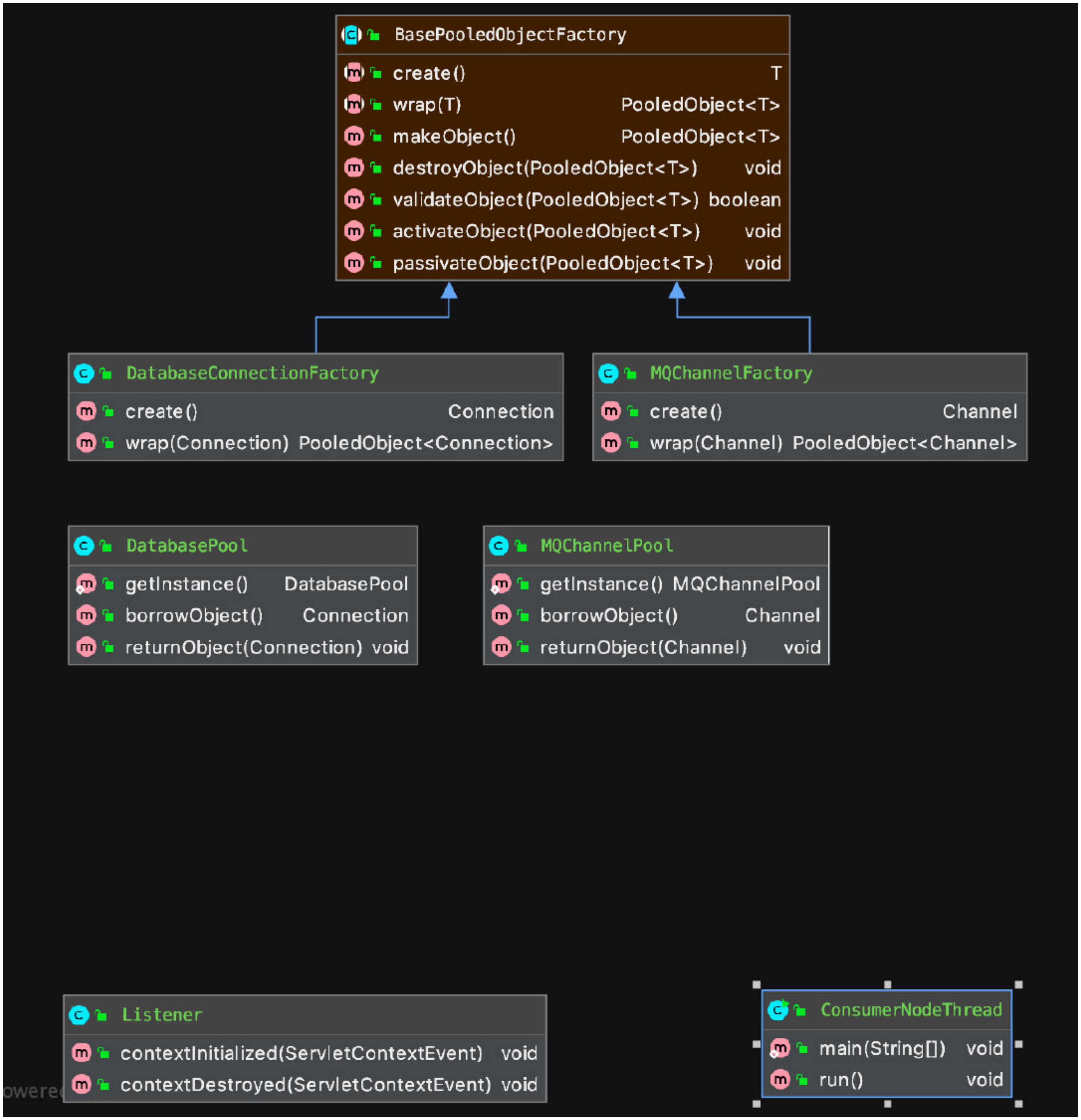
Assignment 3

Github Repo

https://github.com/duskcloudxu/bsds2020fall_Assignment

Project Structure


BasePooledObjectFactory
create() T
wrap(T) PooledObject<T>
makeObject() PooledObject<T>
destroyObject(PooledObject<T>) void
validateObject(PooledObject<T>) boolean
activateObject(PooledObject<T>) void
passivateObject(PooledObject<T>) void
DatabaseConnectionFactory
create() Connection
wrap(Connection) PooledObject<Connection>
MQChannelFactory
create() Channel
wrap(Channel) PooledObject<Channel>
DatabasePool
getInstance() DatabasePool
borrowObject() Connection
returnObject(Connection) void
MQChannelPool
getInstance() MQChannelPool
borrowObject() Channel
returnObject(Channel) void
Listener
contextInitialized(ServletContextEvent) void
contextDestroyed(ServletContextEvent) void
ConsumerNodeThread
main(String[]) void
run() void
owere

New Added Modules

- **Listener**
  - initialization class that would init 10 consumer threads at the setup stage of tomcat server.
- **DatabasePool, MQChannelPool, DatabaseConnectionFactory, MQChannelFactory**
  - In *Singleton* design pattern
  - Pooling mechanism for reusage of expensive resources like database connections and rabbitMQ connections. Greatly Improved the performance
- **ConsumerThreads**
  - Threads that intialized in the setup stage of server, comsure post messages from RabbitMQ and write it to the database.

Workflow With RabbitMQ

- With **RabbitMQ**, we could *split* the database writing and response when we handling the post request.
- In previous practice, we write the data to the database, then return the result to the client, and in the case of large network traffic, it's common to have time-out. In this assignment, we *splited* the database writing and the response. Servlet would only check the validity of writing data, and forward it into the RabbitMQ, and return 203 code to the client. In the meanwhile, consumer threads would take message from the RabbitMQ and write it to the database.
- One thing worth to mention is that we added connection pool in this version and it brought great improvement in the performance, since connection establishment is always expensive.

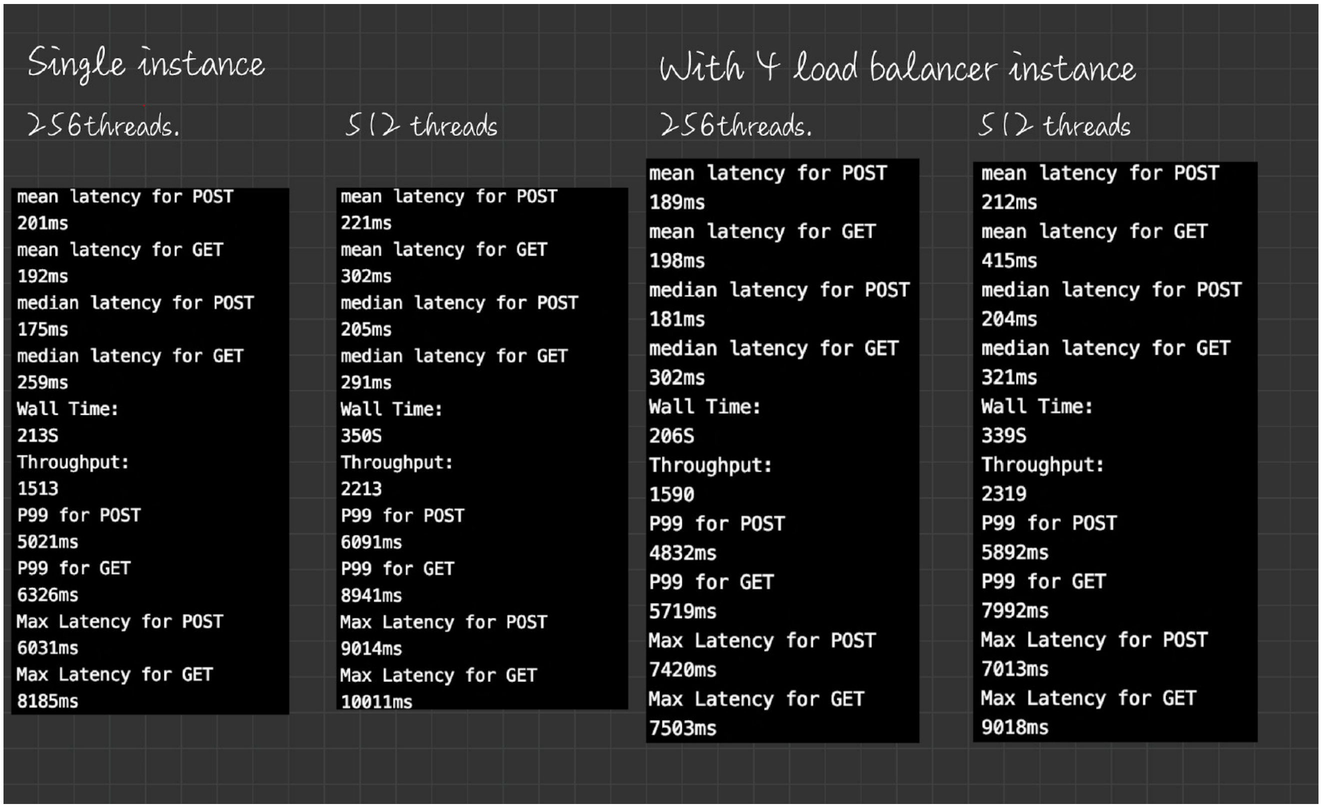
Edge case Handling

- Server or MQ crush
  - Channel is created as persistent type so even if server or MQ was crushed, the message would still remain in the disk and could be retrieved when the server back online.
- Thread load balance
  - Each consumer comes with `channel.basicQos(5);`, which means they would take 5 messages at a time, and the MQ would not assign taks to a comsumer if it does not finish its current tasks.

Performance Comparation

numThread\metric	Mean Post Latency(ms)	Mean Get Latency(ms)	Median Post Latency(ms)	Median Get Latency(ms)	Wall Time(Sec)
256	1151	1506	1222	961	2341

numThread\metric	Mean Post Latency(ms)	Mean Get Latency(ms)	Median Post Latency(ms)	Median Get Latency(ms)	Wall Time(Sec)	Throughput(per Sec)	P99 of Post	P99 of Get	Max Response of Post	Max Response of Get
256 w/o load balancing	201	192	175	259	213	1513	5021	6326	6031	8185
512 w/o load balancing	221	302	205	291	350	2213	6091	8941	9014	10011
256 w/ load balancing	189	198	181	302	206	1590	4821	5719	7420	7503
512 w/ load balancing	212	415	204	321	339	2319	5892	7992	7013	9018



```
mean latency for POST
201ms
mean latency for GET
192ms
median latency for POST
175ms
median latency for GET
259ms
Wall Time:
213S
Throughput:
1513
P99 for POST
5021ms
P99 for GET
6326ms
Max Latency for POST
6031ms
Max Latency for GET
8185ms
```

```
mean latency for POST
221ms
mean latency for GET
302ms
median latency for POST
205ms
median latency for GET
291ms
Wall Time:
350S
Throughput:
2213
P99 for POST
6091ms
P99 for GET
8941ms
Max Latency for POST
9014ms
Max Latency for GET
10011ms
```

```
mean latency for POST
189ms
mean latency for GET
198ms
median latency for POST
181ms
median latency for GET
302ms
Wall Time:
206S
Throughput:
1590
P99 for POST
4832ms
P99 for GET
5719ms
Max Latency for POST
7420ms
Max Latency for GET
7503ms
```

```
mean latency for POST
212ms
mean latency for GET
415ms
median latency for POST
204ms
median latency for GET
321ms
Wall Time:
339S
Throughput:
2319
P99 for POST
5892ms
P99 for GET
7992ms
Max Latency for POST
7013ms
Max Latency for GET
9018ms
```

Analysis

- Pooling and Message queue greatly improved the server performance, and it made the load balancing an optional choice. We can see from the comparison above that the load balancing did increase the performance but only in a slight scale. However, there should be a better structure, which is to use one powerful instance as the MQ instance and some other instance as the consumer, the consumer cluster could be a elastic group in AWS so it would be flexible enough according to the network traffic situation.

Exploration and answers

- Do I need load balancing? Or can my system work with 1 free-tier (or slightly upgraded) server
    - if your instance are power enough(i.e. an t3.large instance), you might do not need load balancing. (MQ did the load balancing for you, at the cost of late database writing)
- How many consumers nodes do I need?
    - I used 10 consumer threads, and increasing consumer threads would not improve the performance on the ack rate of MQ.

P.S.

In order to prevent costing too much aws credits, I will shutdown all the extra ec2 instance in the cluster after I submit this report, so please let me know if you want to test my server performance.