the URL for your git repo.

https://github.com/jerryyummy/distributed_system_2(for client)
https://github.com/jerryyummy/assignment2_server

description of your server design. Include major classes, packages, relationships, how messages get sent/received, etc

1. Consumer

Imports:
It uses Gson, a popular Java library for converting Java Objects into their JSON representation and vice versa.
There are several imports from com.rabbitmq.client, indicating interaction with RabbitMQ, a message broker.
It uses Apache Commons Pool, which is typically used for pooling objects like database connections.
There are SQL-related imports (java.sql.DriverManager, java.sql.PreparedStatement, java.sql.SQLException), suggesting database interaction.
Class Declaration: The class Consumer is declared without extending any other class or implementing interfaces (at least in the part visible).
Fields: It defines constants and fields like QUEUE_NAME and JDBC_USER, which are likely used for RabbitMQ queue identification and database access, respectively.

JDBC Connection: The class sets up a JDBC connection to a MySQL database, as indicated by the JDBC URL, user, and password fields. The database seems to be an Amazon RDS instance (database-1.cxbykd0hqw1f.us-west-2.rds.amazonaws.com), used to store album-related data.
RabbitMQ Connection: It establishes a connection to a RabbitMQ server (factory.setHost("54.184.133.208")). The RabbitMQ server is expected to be running on port 5672 with the given credentials.

Queue Declaration: It declares a queue (QUEUE_NAME) for message consumption from RabbitMQ.
Message Consumption: The class implements a consume() method that sets up a DeliverCallback. This callback is triggered upon receiving messages from the RabbitMQ queue.
Database Interaction: Upon receiving a message, the Consumer inserts data into the database. Specifically, it performs an INSERT operation into a table named rate, creating records with fields like person, islike, and title.
Error Handling: There is basic error handling for exceptions during database operations, with a try-catch-finally block.

First, create a factory and a connection, then create a channel from the connection. Second, create a queue to receive message from producer, once received, just parse the message to get the album item info. Third, use jdbc to access database and insert records, close the connection. Finally, I run it in my Main class and make it run constantly so that it can receive message without creating new consumers.

2. Producer

Imports: The class imports RabbitMQ client libraries, indicating its role in message queue operations. There's also an import from org.apache.commons.pool2.impl.GenericObjectPool, likely for managing pooled resources.
Class Declaration: The class Producer is defined with a constructor that accepts a Connection object.
Fields:
QUEUE_NAME: This constant matches the queue name in Consumer.java, suggesting that both classes interact with the same RabbitMQ queue.

connection: A Connection object is maintained, presumably a RabbitMQ connection.
Methods:
produce(String message): This method, though not fully visible, is evidently designed to send messages to the RabbitMQ queue.

Queue Declaration: Similar to Consumer.java, it declares a queue with the name QUEUE_NAME ("hello"). This indicates that both the producer and consumer are working with the same queue in RabbitMQ.
Message Production: The produce(String message) method:
Creates a Channel using the provided Connection object.
Declares the queue (ensuring its existence before sending messages).
Publishes the message to the queue using channel.basicPublish. The message is converted into bytes (message.getBytes()).
Outputs a confirmation to the console indicating that the message has been sent.
Closes the channel after sending the message.

First, create a factory and a connection in servlet.init() method so that we can reuse the connection. Second, name a queue to produce message, once we post a request, create a new channel just parse the request to get the album item info and insert it into the queue. Finally, I run it in my Servlet class and make it run every time when I request the servlet.

3. Servlet
Imports:
The Gson library for JSON processing.
RabbitMQ classes (ConnectionFactory), indicating interaction with message queues.
Various org.example.bean classes (Album, Profile, Status), likely representing data structures.
Classes from org.example.queue (Consumer, Producer), suggesting integration with the message queueing system.
Standard Java Servlet API classes and annotations, such as HttpServlet, HttpServletRequest, ServletException, and MultipartConfig.
Servlet Configuration:
The use of servlet-related imports and annotations indicates that this class is a servlet, likely extending HttpServlet.
The MultipartConfig annotation suggests handling of multipart/form-data requests, commonly used for file uploads.
From this overview, it appears that AlbumReviewServlet is a servlet designed to handle HTTP requests, possibly related to album reviews. It seems to integrate with the RabbitMQ messaging system, likely for sending or receiving messages related to album reviews.

To complete the analysis, I'll delve further into the file to understand its core functionalities, such as how it processes HTTP requests and interacts with other components of your server design. Let's proceed with a deeper examination.

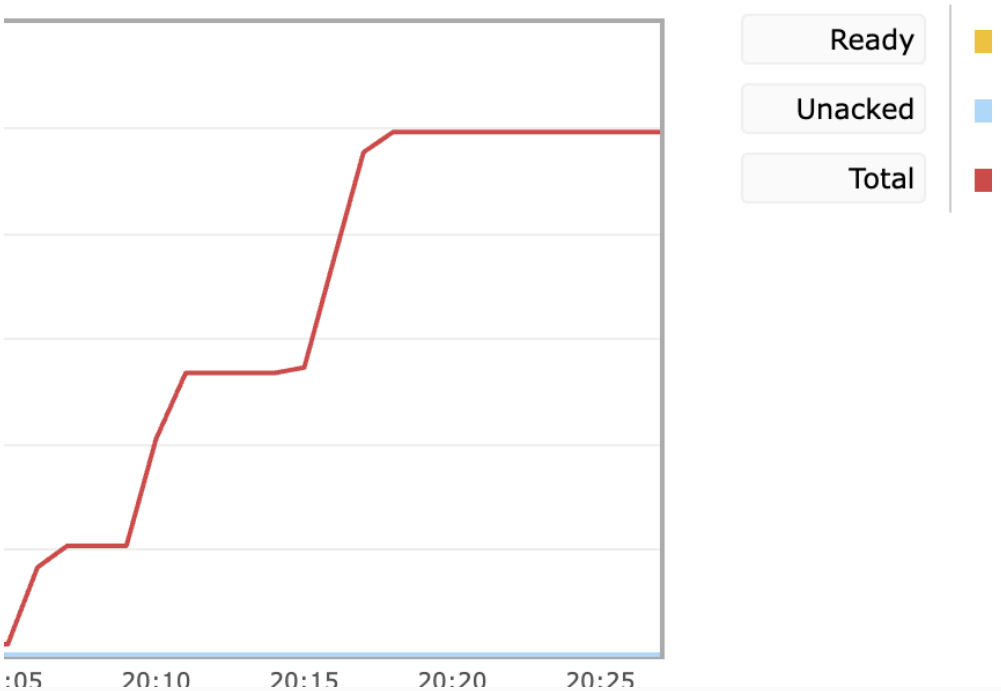
Detailed Analysis of AlbumReviewServlet.java
The AlbumReviewServlet.java file has several key features:

Servlet Configuration:
It extends HttpServlet, making it a Java servlet capable of handling HTTP requests.
The @MultipartConfig annotation configures it to handle multipart/form-data requests, suitable for file uploads.
Fields:

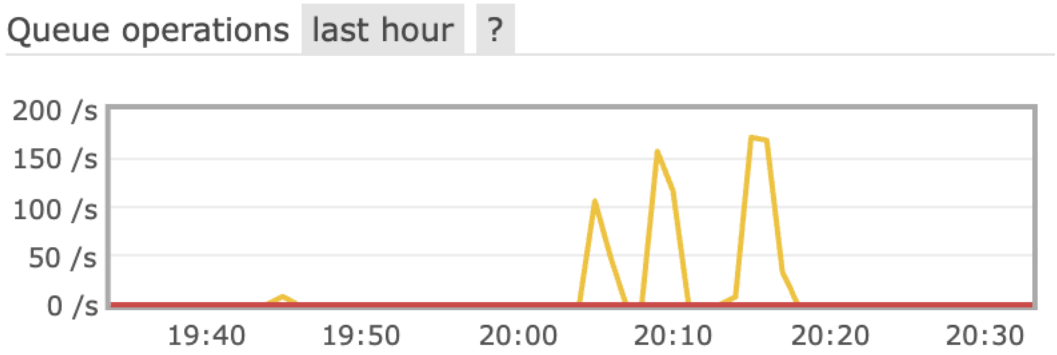
Producer producer: An instance of Producer, likely used to send messages to the RabbitMQ queue.
JDBC credentials and URL: Similar to Consumer.java, it connects to the same MySQL database.
QUEUE_NAME: This constant matches the queue name used in both Producer.java and Consumer.java, indicating interactions with the same RabbitMQ queue.
connection: A field for database connection.
HOST: The RabbitMQ server address, consistent with other classes.

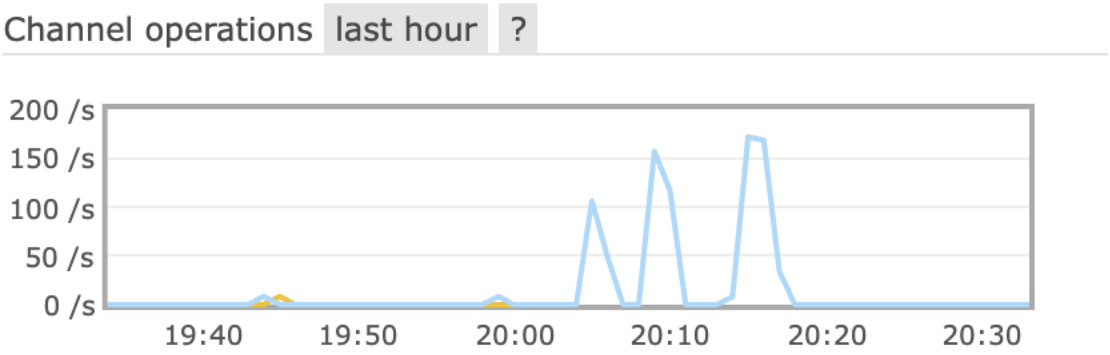
Initialization (init method):
The method likely establishes connections to both the RabbitMQ server and the database, so that we can reuse
Post():
Get the info from request, and we invoke producer to send message
Destroy():
We close the connection with rabbitmq to save resource.

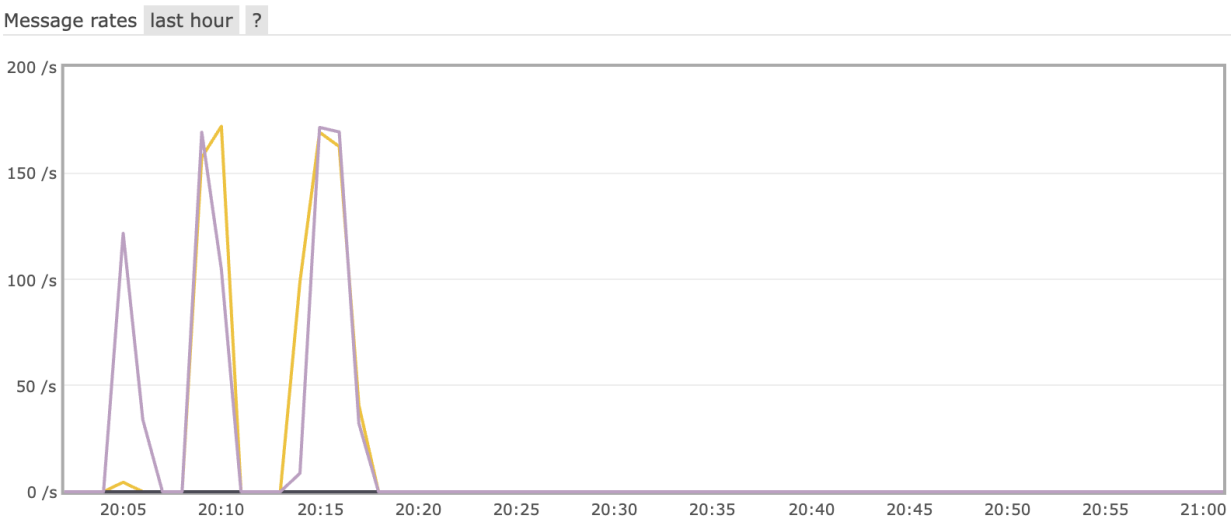
Test run results (command lines showing metrics, RMQ management windows showing queue size, send/receive rates) showing your best throughput.



It shows that the queue size will increase to a plateau and maintain and after message has been sent it will decrease to 0;







After three tests, we can see that the queue and channel will form a plateau, and the rates is about 170/s

```
for post:
mean response time: 208
median response time: 168
99% response time: 67
min response time: 47
max response time: 9563
succeed request:18673
for post:
mean response time: 378
median response time: 249
99% response time: 70
min response time: 48
max response time: 12383
succeed request:32471
for post:
mean response time: 536
median response time: 283
99% response time: 72
min response time: 42
max response time: 42261
succeed request:44548
```

	runtime	throughput	min	max	mean
10,10,2	55425ms	297	47	9563	208
10,20,2	98706ms	304	48	12383	378
10,30,2	143749ms	322	42	42261	526

Queues

All queues (1)

Overview					Messages			Message rat
Virtual host	**Name**	**Type**	Features	**State**	**Ready**	**Unacked**	**Total**	**incoming**
/	**hello**	classic		running	53,866	0	53,866	194/s

Add a new queue