Project

Report on

Message-Driven Data Processing with RabbitMQ and MongoDB

Submitted to

Prof. Yan Liu

COEN 6731
Distributed Software Systems

Department of Electrical and Computer Engineering
Gina Cody School of Engineering and Computer Science

By

Pranav Kumar Jha

Student ID: 40081750

18th April, 2023

Introduction

This project involves the development of a distributed data processing system for analyzing educational cost and economic data in the United States. The system is built using a microservice architecture and message queuing with RabbitMQ. The data is stored in MongoDB, and the application is developed using Java. It is using the RabbitMQ Java client library and the MongoDB Java driver to connect to a RabbitMQ message broker and a MongoDB database respectively.

The system is composed of two main components: a producer and a consumer. The producer is responsible for generating messages in response to user queries, and the consumer is responsible for processing these messages and producing responses. The producer and consumer communicate through a RabbitMQ message broker, which enables asynchronous communication and decouples the two components.

The producer receives user queries and generates messages containing the necessary parameters for data retrieval and processing. The messages are then published to a RabbitMQ exchange using a topic-based routing key, which enables selective message routing to the appropriate queue(s).

The consumer subscribes to the appropriate queues and receives messages for processing. The messages are parsed and the necessary data is retrieved from MongoDB. The consumer then generates a response message and publishes it to the exchange with the appropriate routing key, allowing the response to be selectively routed to the appropriate destination(s).

The system supports five different queries, each with their own message format and response format. The queries involve retrieving data on educational cost, economic indicators, and growth rates, and are designed to provide insights into trends and patterns in the data.

Data Operation Architecture

The architecture diagram shown in figure below is depicting the flow of data between a database, a producer, and multiple consumers using an exchange and multiple queues.

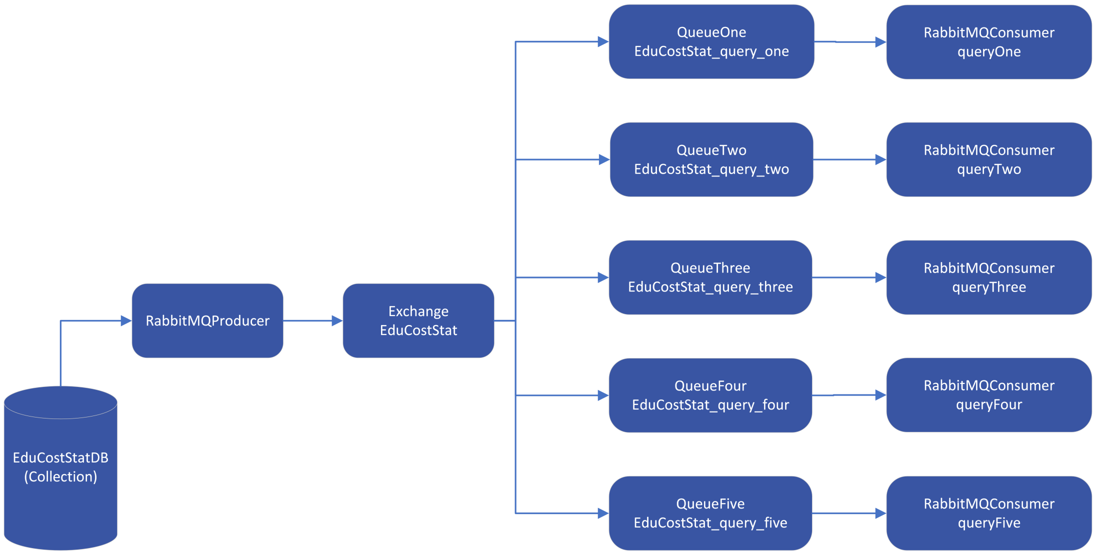

Fig. 1: Message oriented architecture

1. The mongoDB database contains collections with educational cost data that can be queried.
2. The producer retrieves data from the database based on a configuration file that specifies parameters for each query, and publishes the data to an exchange topic.
3. The exchange has multiple queues bound to it, each with a different routing key based on the parameters of the query.
4. The queues are named according to the topic they are bound to (e.g. EduCostStat_query_one, EduCostStat_query_two, etc.) and store messages received from the producer until they are consumed by a consumer.
5. The consumer subscribes to one or more queues, depending on the topics they are interested in, and receives messages from the queues based on the routing key used by the producer. The consumer processes the messages and displays the data to the user or prints it to the console.

1 Task 1: Installing rabbitmq server on the local computer

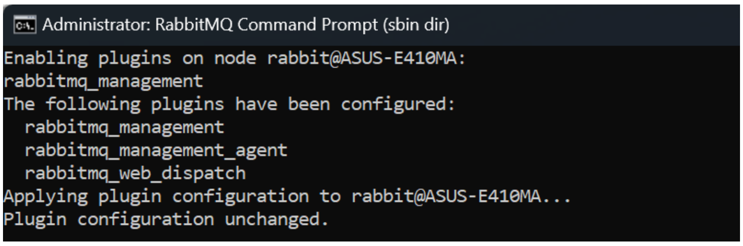

Figure 1.1: RabbitMQ management console

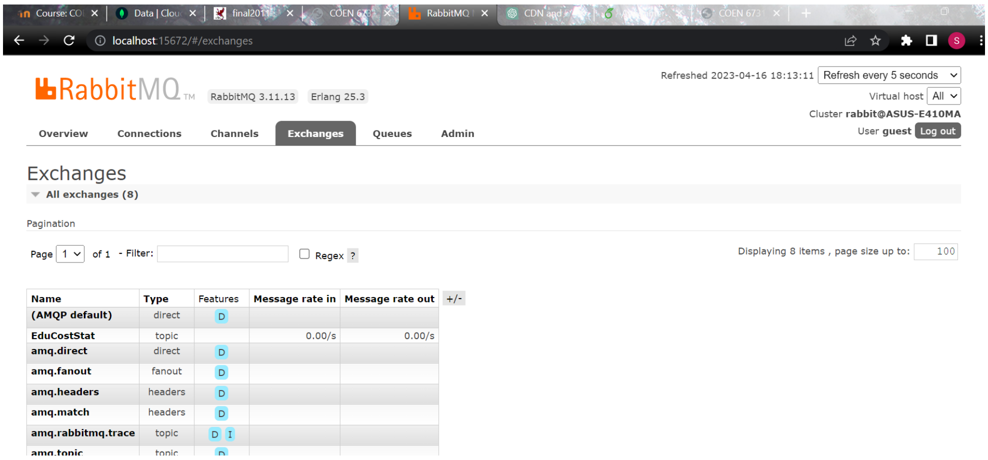

Figure 1.2: RabbitMQ localhost user interface

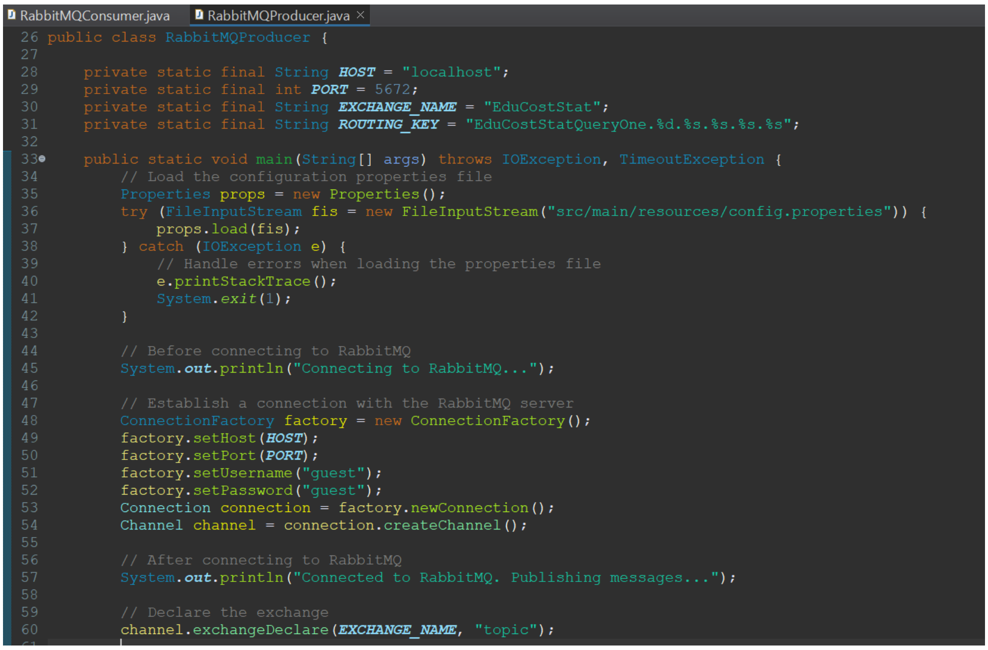

Figure 1.3: Producer connecting to RabbitMQ

This Java class, RabbitMQProducer, is responsible for publishing messages to a RabbitMQ message broker. It establishes a connection with the RabbitMQ server, declares a topic exchange named ”EduCostStat”, and uses the routing key ”EduCostStatQueryOne.%d.%s.%s.%s.%s” to specify the routing of messages to queues. The class loads the configuration properties from a file and sets up the connection parameters, such as the host and port, and the username and password. Once the connection is established, the class publishes messages to the exchange through the channel. The messages are sent in a topic format, which allows them to be delivered to specific queues based on their routing key.

Collection	**Parameters**	**Topic**
EduCostStatQueryOne	Query the cost given specific year, state, type, length, expense	Cost-[Year]-[State]-[Type]-[Length]
EduCostStatQueryTwo	Query the top 5 most expensive states (with overall expense) given a year, type, length	Top5-Expensive-[Year]-[Type]-[Length]
EduCostStatQueryThree	Query the top 5 most economic states (with overall expense) given a year, type, length	Top5-Economic-[Year]-[Type]-[Length]
EduCostStatQueryFour	Query the top 5 states of the highest growth rate of overall expense given a range of past years, one year, three years and five years (using the latest year as the base), type and length	Top5-HighestGrow-[Years]
EduCostStatQueryFive	Aggregate region‘s average overall expense for a given year, type and length	AverageExpense-[Year]-[Type]- [Length]

Table 2.1: Data collection generated from Assignment II

2 Task 2: Programming the producer and consumer using rabbitmq exchange topic libraries

2.1 RabbitMQProducer

2.1.1 Customized Data Retrieval from MongoDB Cloud Service

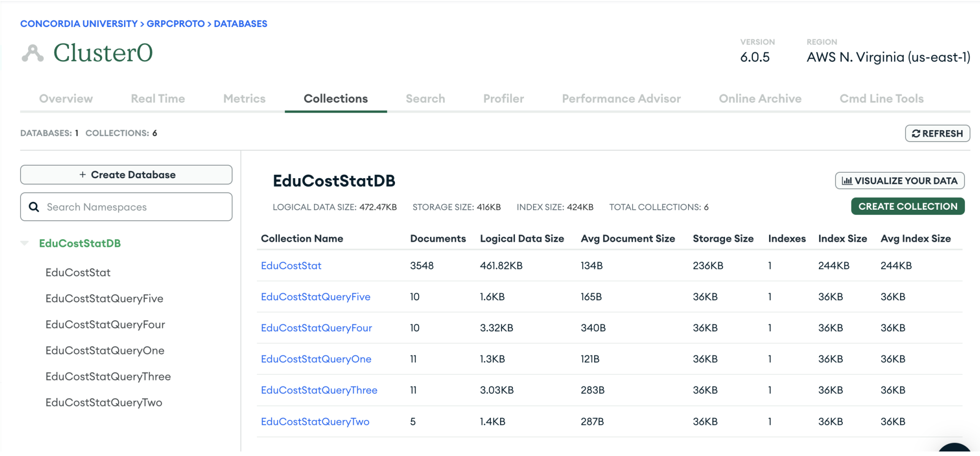

Figure 2.1: Collections generated in the MongoDB Cluster from Assignment II

The producer retrieves the datasets from each collection from the MongoDB cloud service for each topic listed in Table 2.1. The parameters to customize each topic is set in a configuration file named "config.properties" as shown in the Figure 2.2 below.

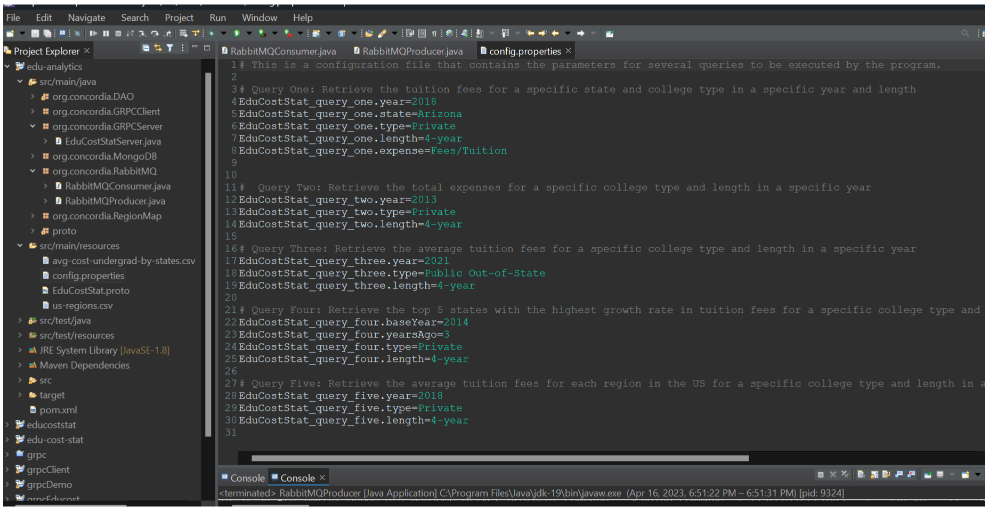

Figure 2.2: Config.properties file

2.1.2 Publishing Data using RabbitMQ Topic Exchanges

The RabbitMQ producer is responsible for creating requests for query data and publishing the data to the exchange topics with routing keys that match the topics for each queue. This enables the consumer to subscribe to specific topics and receive only the messages that are relevant to their interests. By routing the messages based on their topics, RabbitMQ ensures that the messages are delivered to the correct consumers efficiently and reliably.

Each queue is typically bound to the exchange with a specific routing key that corresponds to the query that the queue is responsible for. The routing key is constructed based on the query parameters and is unique to that particular query.

So, when a producer publishes a message to the exchange with a routing key that matches the routing key for a particular query, the message will be routed to the queue that is bound to that exchange with that routing key, which is the queue responsible for handling that particular query. This ensures that each message is received by the appropriate consumer based on the query that it relates to, regardless of the number of queues or queries being used.

Avoiding Multithreading for Independent Producer and Consumer Operations:

The producer and consumer components are designed to operate on the same node without utilizing multi-threading within the same application. This constraint ensures that each component runs independently, facilitating efficient and reliable communication between them through the RabbitMQ message broker, while maintaining the overall system's performance and stability.

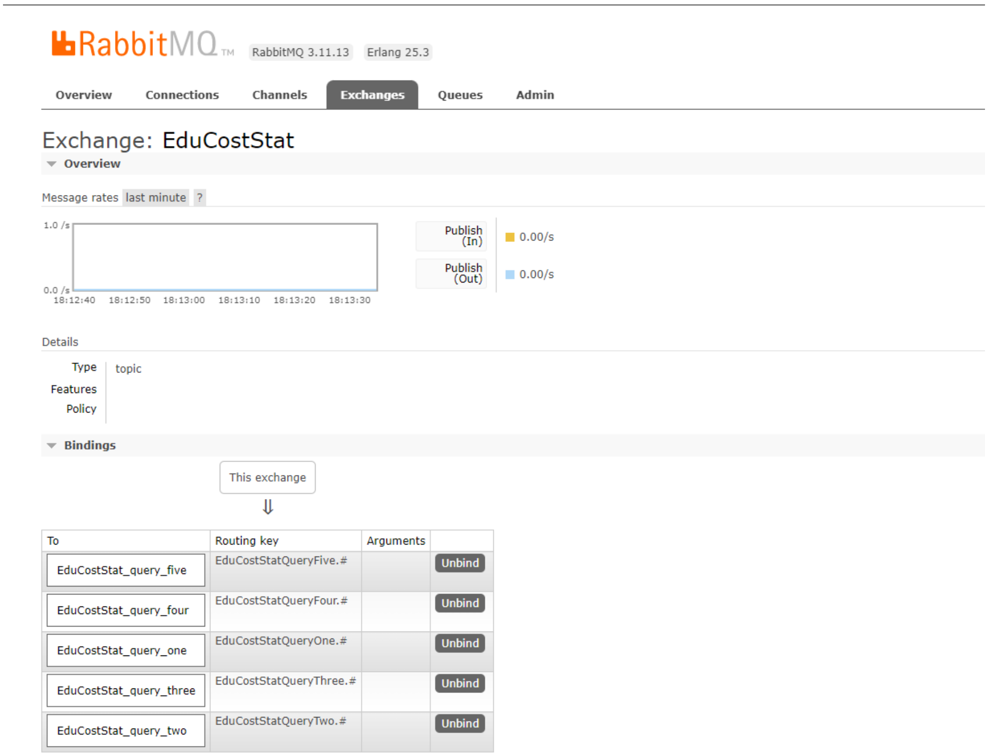

Figure 2.3: Routing Key and Queue Binding for each queue

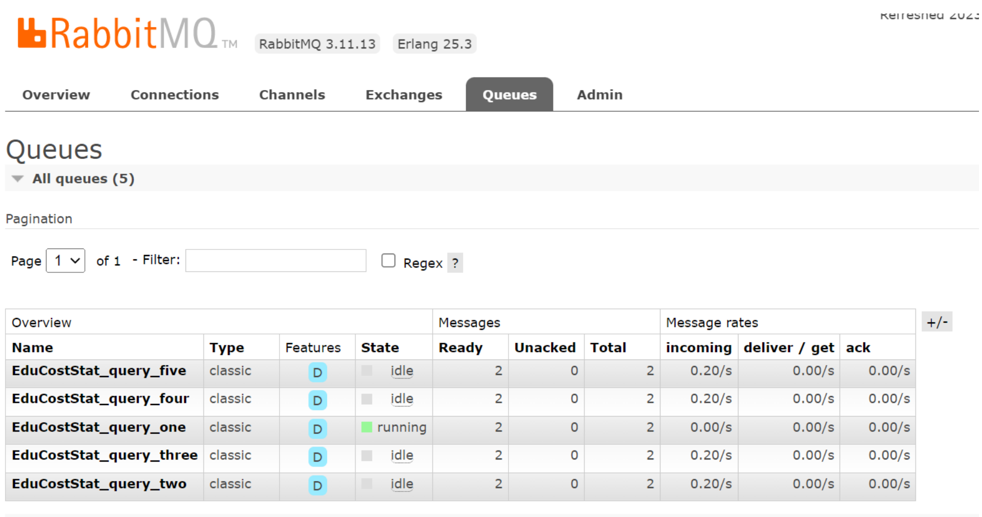

Figure 2.4: Messages sent by the producer are received and stored in the corresponding RabbitMQ queue, awaiting consumption by the consumer

```java
// Query One
// Create a QueryOneRequest object with the specified properties from the configuration file
QueryOneRequest queryOneRequest = QueryOneRequest.newBuilder()
        .setYear(Integer.parseInt(props.getProperty("EduCostStat_query_one.year")))
        .setState(props.getProperty("EduCostStat_query_one.state"))
        .setType(props.getProperty("EduCostStat_query_one.type"))
        .setLength(props.getProperty("EduCostStat_query_one.length"))
        .setExpense(props.getProperty("EduCostStat_query_one.expense")).build();

// Serialize the message to a byte array
byte[] messageBytes = queryOneRequest.toByteArray();

// Set the routing key
String routingKey = String.format(ROUTING_KEY, queryOneRequest.getYear(), queryOneRequest.getState(),
        queryOneRequest.getType(), queryOneRequest.getLength(), queryOneRequest.getExpense());

// Publish the message to the exchange with the appropriate routing key
channel.basicPublish(EXCHANGE_NAME, routingKey, null, messageBytes);
System.out.println("Sent message for query one: " + queryOneRequest);

// Get the collection from MongoDB
MongoCollection<Document> collection = MongoDBUtil.getCollection("EduCostStatQueryOne");

// Retrieve the query result from MongoDB
List<Object> queryArray = Arrays.asList(queryOneRequest.getYear(), queryOneRequest.getState(),
        queryOneRequest.getType(), queryOneRequest.getLength(), queryOneRequest.getExpense());
Document result = collection.find(Filters.eq("query", queryArray)).first();

// Print the query result to the console
if (result != null) {
    Integer expense = result.getInteger("result");
    System.out.println("Query One result: " + expense);
} else {
    System.out.println("Query result is not found in the collection.");
}
```

Figure 2.5: Creating Request for Query One to publish the data to the exchange topics with a routing key that matches the topic for queue one.

In our project, the RabbitMQProducer Java class is responsible for handling messaging related to Query One. The code block above shows how the class creates a QueryOneRequest object, serializes it, and sends it to the RabbitMQ broker.

To create the QueryOneRequest object, the code reads the required values from the configuration file and sets them using the QueryOneRequest builder. This builder creates an instance of QueryOneRequest that contains the year, state, type, length, and expense values required to perform Query One.

The next step in the code is to serialize the QueryOneRequest object into a byte array. This is necessary to send the object as a message to the RabbitMQ broker.

The code then sets the routing key for the message using the values from the QueryOneRequest object. The routing key is used to ensure that the message is delivered to the correct queue in the broker.

After setting the routing key, the code publishes the message to the RabbitMQ broker using the basicPublish method. The message contains the serialized QueryOneRequest object, the exchange name, and the routing key.

The next section of the code retrieves the query result from MongoDB. It retrieves the query result by querying the MongoDB collection using the query information stored in the Query-

OneRequest object.

If the query result is found, the code prints it to the console. Otherwise, it outputs a message indicating that the query result was not found in the collection.

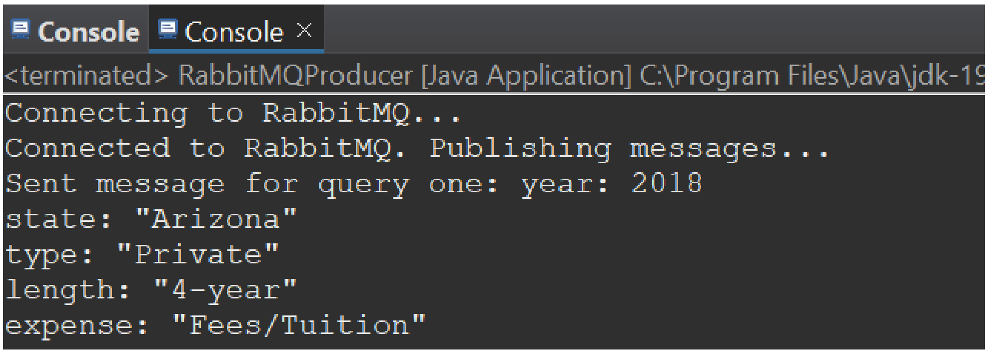

Figure 2.6: Producer publishes the message for query one

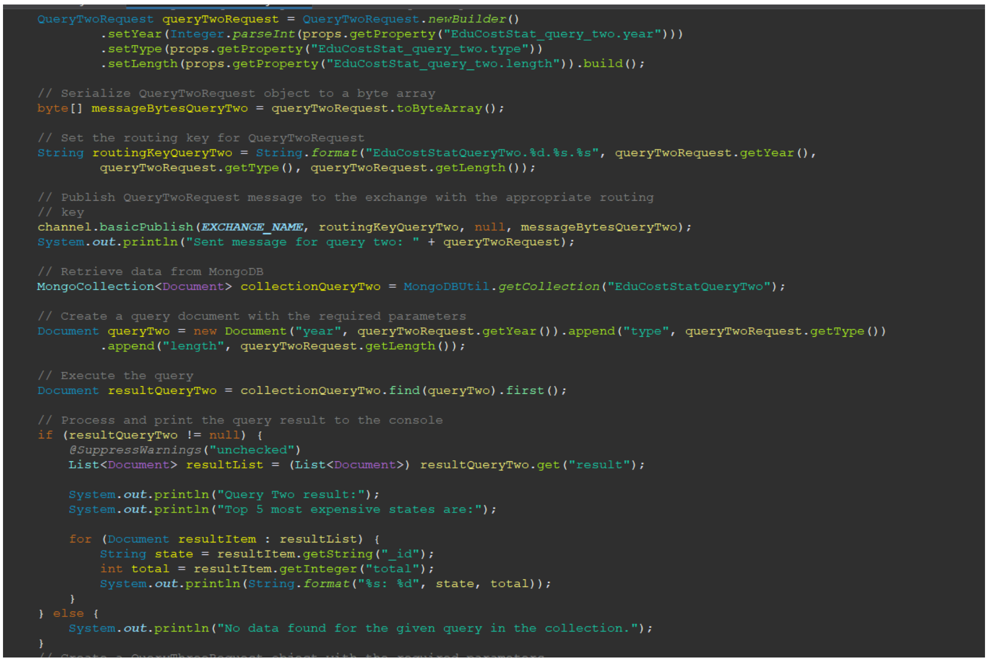

Figure 2.7: Creating Request for Query Two to publish the data to the exchange topics with a routing key that matches the topic for queue Two.

The above code block shows how Query Two requests are handled in our project using the RabbitMQProducer Java class. Query Two is a more complex query that involves retrieving data from MongoDB and then processing and printing the result.

The first section of the code creates a QueryTwoRequest object with the year, type, and length values read from the configuration file. This object is then serialized to a byte array and the routing key is set using the year, type, and length values from the QueryTwoRequest object. The message containing the serialized QueryTwoRequest object is then published to the RabbitMQ broker using the appropriate routing key.

After sending the message, the code retrieves data from MongoDB by executing a query on the EduCostStatQueryTwo collection. The query is constructed using the year, type, and length values from the QueryTwoRequest object.

The code then processes the query result and prints it to the console. If the query returns a result, the code prints the top 5 most expensive states and their total expenses. If there is no data found for the given query in the collection, the code outputs a message indicating that no data was found.

```
Query Two result:
Top 5 most expensive states are:
Massachusetts: 49871
District of Columbia: 48440
Connecticut: 48262
Vermont: 46255
Rhode Island: 46114
Sent message for query three: year: 2021
type: "Public Out-of-State"
length: "4-year"
```

Figure 2.8: Producer publishes the message for query two

```java
QueryThreeRequest queryThreeRequest = QueryThreeRequest.newBuilder()
        .setYear(Integer.parseInt(props.getProperty("EduCostStat_query_three.year")))
        .setType(props.getProperty("EduCostStat_query_three.type"))
        .setLength(props.getProperty("EduCostStat_query_three.length")).build();

// Serialize the message to a byte array
byte[] messageBytesQueryThree = queryThreeRequest.toByteArray();

// Set the routing key for QueryThreeRequest
String routingKeyQueryThree = String.format("EduCostStatQueryThree.%d.%s.%s", queryThreeRequest.getYear(),
        queryThreeRequest.getType(), queryThreeRequest.getLength());

// Publish the message to the exchange with the appropriate routing key
channel.basicPublish(EXCHANGE_NAME, routingKeyQueryThree, null, messageBytesQueryThree);
System.out.println("Sent message for query three: " + queryThreeRequest);

// Retrieve data from MongoDB
MongoCollection<Document> collectionQueryThree = MongoDBUtil.getCollection("EduCostStatQueryThree");

// Create a query document with the required parameters
Document queryThree = new Document("year", queryThreeRequest.getYear())
        .append("type", queryThreeRequest.getType()).append("length", queryThreeRequest.getLength());

// Execute the query
List<Document> resultListQueryThree = collectionQueryThree.find(queryThree).into(new ArrayList<>());

if (!resultListQueryThree.isEmpty()) {
    Document resultItem = resultListQueryThree.get(0);
    @SuppressWarnings("unchecked")
    List<Document> results = (List<Document>) resultItem.get("result");

    if (results != null) {
        System.out.println("Query Three result:");
        System.out.println("Top 5 most economic states are:");
        for (Document doc : results) {
            String state = doc.getString("_id");
            Integer total = doc.getInteger("total");
            if (total != null) {
                System.out.println(String.format("%s: %d", state, total.intValue()));
            } else {
                System.out.println(String.format("%s: null", state));
            }
        }
```

Figure 2.9: Creating Request for Query Three to publish the data to the exchange topics with a routing key that matches the topic for queue Three.

The code block above is part of the RabbitMQProducer Java class and is responsible for handling Query Three requests. Query Three involves retrieving data from MongoDB and printing the top 5 most economic states.

The first section of the code creates a QueryThreeRequest object with the year, type, and length values from the configuration file. The QueryThreeRequest object is then serialized to a byte array and the routing key is set using the year, type, and length values from the QueryThreeRequest object. The message containing the serialized QueryThreeRequest object is then published to the RabbitMQ broker using the appropriate routing key.

After sending the message, the code retrieves data from MongoDB by executing a query on the EduCostStatQueryThree collection. The query is constructed using the year, type, and length values from the QueryThreeRequest object.

The code then processes the query result and prints it to the console. If the query returns a result, the code prints the top 5 most economic states and their total expenses. If there is no data found for the given query in the collection, the code outputs a message indicating that no data was found.

```
Query Three result:
Top 5 most economic states are:
District of Columbia: 13004
South Dakota: 21090
North Dakota: 22493
Wyoming: 24509
Florida: 29325
Sent message for query four: baseYear: 2014
yearsAgo: 3
type: "Private"
length: "4-year"
```

Figure 2.10: Producer publishes the message for query three

```java
QueryFourRequest queryFourRequest = QueryFourRequest.newBuilder()
        .setBaseYear(Integer.parseInt(props.getProperty("EduCostStat_query_four.baseYear")))
        .setYearsAgo(Integer.parseInt(props.getProperty("EduCostStat_query_four.yearsAgo")))
        .setType(props.getProperty("EduCostStat_query_four.type"))
        .setLength(props.getProperty("EduCostStat_query_four.length")).build();

byte[] messageBytesQueryFour = queryFourRequest.toByteArray();
String routingKeyQueryFour = String.format("EduCostStatQueryFour.%d.%d.%s.%s", queryFourRequest.getBaseYear(),
        queryFourRequest.getYearsAgo(), queryFourRequest.getType(), queryFourRequest.getLength());

// Then, we publish the message to the RabbitMQ exchange.

channel.basicPublish(EXCHANGE_NAME, routingKeyQueryFour, null, messageBytesQueryFour);
System.out.println("Sent message for query four: " + queryFourRequest);

// Next, we retrieve a MongoDB collection, create a query document with the
// required parameters, and execute the query.

MongoCollection<Document> collectionQueryFour = MongoDBUtil.getCollection("EduCostStatQueryFour");

Document queryFour = new Document("query", Arrays.asList(queryFourRequest.getBaseYear(),
        queryFourRequest.getYearsAgo(), queryFourRequest.getType(), queryFourRequest.getLength()));

FindIterable<Document> iterable = collectionQueryFour.find(queryFour);

// Finally, we iterate over the query results and retrieve the top 5 states with
// the highest growth rate.

System.out.println("Query Four result:");
for (Document doc : iterable) {
    @SuppressWarnings("unchecked")
    List<Document> results = (List<Document>) doc.get("result");
    System.out.println("Top 5 states of highest growth rate are:");
    if (results != null) {
        for (Document result1 : results) {
            String state = result1.getString("state");
            Double growthRate = result1.getDouble("growthRate");
            System.out.println(String.format("%s: %.2f", state, growthRate));
        }
    }
}
// In this code snippet, we are sending a message to a RabbitMQ exchange and
```

Figure 2.11: Creating Request for Query Four to publish the data to the exchange topics with a routing key that matches the topic for queue Four.

The code block above is part of the RabbitMQProducer Java class and is responsible for handling Query Four requests. Query Four involves retrieving data from MongoDB and printing the top 5 states with the highest growth rate.

The first section of the code creates a QueryFourRequest object with the base year, years ago, type, and length values from the configuration file. The QueryFourRequest object is then serialized to a byte array and the routing key is set using the base year, years ago, type, and length values from the QueryFourRequest object. The message containing the serialized QueryFourRequest object is then published to the RabbitMQ broker using the appropriate routing key.

After sending the message, the code retrieves data from MongoDB by executing a query on the EduCostStatQueryFour collection. The query is constructed using the base year, years ago, type, and length values from the QueryFourRequest object.

The code then processes the query result and prints it to the console. The code iterates over the query results and retrieves the top 5 states with the highest growth rate. If there is no data found for the given query in the collection, the code outputs a message indicating that no data was found.

```
Top 5 states of highest growth rate are:
Idaho: 7.56
Wyoming: 7.52
Iowa: 3.38
New Mexico: 3.08
South Dakota: 2.23
Sent message for query five: year: 2018
type: "Private"
length: "4-year"
```

Figure 2.12: Producer publishes the message for query four

```java
QueryFiveRequest queryFiveRequest = QueryFiveRequest.newBuilder()
        .setYear(Integer.parseInt(props.getProperty("EduCostStat_query_five.year")))
        .setType(props.getProperty("EduCostStat_query_five.type"))
        .setLength(props.getProperty("EduCostStat_query_five.length")).build();

byte[] messageBytesQueryFive = queryFiveRequest.toByteArray();
String routingKeyQueryFive = String.format("EduCostStatQueryFive.%d.%s.%s", queryFiveRequest.getYear(),
        queryFiveRequest.getType(), queryFiveRequest.getLength());

// Next, we retrieve a MongoDB collection, create a query document with the
// required parameters, and execute the query.

MongoCollection<Document> collectionQueryFive = MongoDBUtil.getCollection("EduCostStatQueryFive");

Document queryFive = new Document("year", queryFiveRequest.getYear()).append("type", queryFiveRequest.getType())
        .append("length", queryFiveRequest.getLength());

Document resultItem = collectionQueryFive.find(queryFive).first();

// Then, we publish the message to the RabbitMQ exchange.

channel.basicPublish(EXCHANGE_NAME, routingKeyQueryFive, null, messageBytesQueryFive);
System.out.println("Sent message for query five: " + queryFiveRequest);
// Finally, we retrieve the query results and print them to the console.

if (resultItem != null) {
    Object result1 = resultItem.get("result");
    if (result1 instanceof Document) {
        Document resultDoc = (Document) result1;
        System.out.println("Query Five result:");
        for (String region : resultDoc.keySet()) {
            Double cost = resultDoc.getDouble(region);
            if (cost != null) {
                System.out.println(String.format("%s: %.2f", region, cost.doubleValue()));
            } else {
                System.out.println(String.format("%s: null", region));
            }
        }
    } else {
        System.out.println("No data found for the given query in the collection.");
    }
}
```

Figure 2.13: Creating Request for Query Five to publish the data to the exchange topics with a routing key that matches the topic for queue Five.

The code snippet shows the implementation of Query Five in the project. First, a QueryFiveRequest object is created with the required parameters, and it is serialized to a byte array. The routing key for the message is set based on the parameters of the query.

Next, a MongoDB collection ”EduCostStatQueryFive” is retrieved, and a query document is created with the required parameters. The query is then executed to retrieve the query results.

After that, the message is published to the RabbitMQ exchange with the appropriate routing key. Finally, the query results are retrieved and printed to the console.

If the query results are not null, they are processed and printed to the console. The results show the cost of education for each region in the given year, type, and length of education. If the query results are null, a message is displayed indicating that no data was found for the given query in the collection.

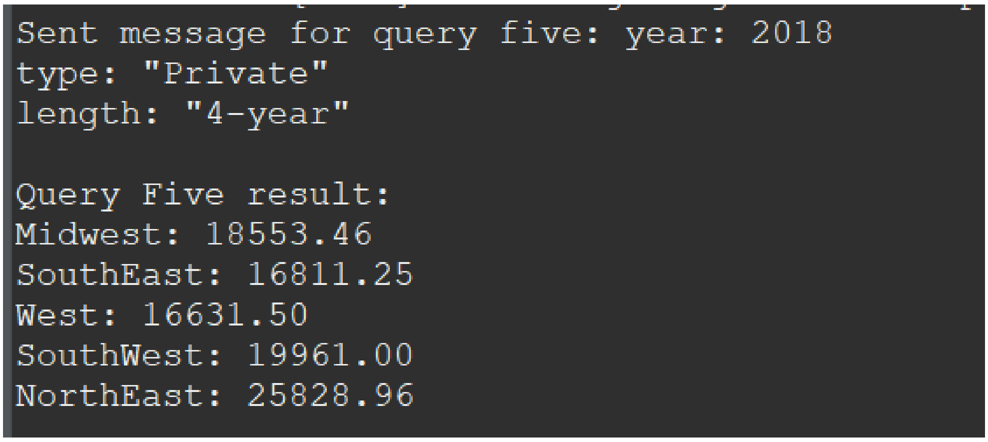

Figure 2.14: Producer publishes the message for query five

2.2 RabbitMQConsumer

The consumer subscribes to a particular topic and receives the data from the corresponding queue associated with that topic.

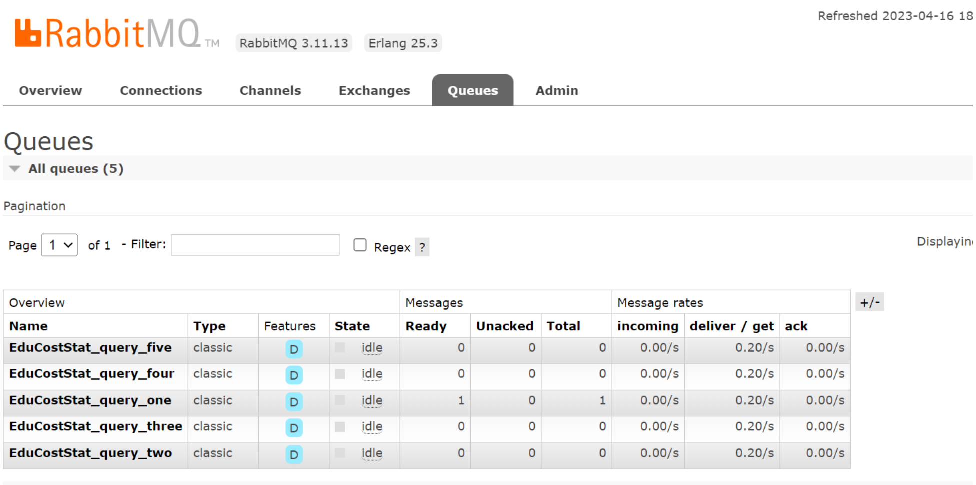

Figure 2.15: Consumer side message consumption. After connecting to the RabbitMQ server, the consumer class starts consuming messages from the queue that it has subscribed to.

```java
public class RabbitMQConsumer {

    // Define constants for RabbitMQ server connection details and exchange name
    private static final String HOST = "localhost";
    private static final int PORT = 5672;
    private static final String EXCHANGE_NAME = "EduCostStat";

    // Method to bind a queue to a specific routing key on a channel
    private static void bindQueueToTopic(Channel channel, String queueName, String routingKey) throws IOException {
        channel.queueBind(queueName, EXCHANGE_NAME, routingKey);
    }

    // Main method that sets up the connection to RabbitMQ and creates and binds
    // queues to specific routing keys
    public static void main(String[] args) throws Exception {

        // Set up connection factory with server details
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost(HOST);
        factory.setPort(PORT);
```

Figure 2.16: Creating the Consumer class and connecting to RabbitMQ localhost

The RabbitMQConsumer class is responsible for consuming messages from RabbitMQ for the EduCostStat exchange. It sets up a connection factory with the RabbitMQ server details and creates and binds queues to specific routing keys.

The class has a constant for the RabbitMQ server's hostname, port, and exchange name. It also has a method bindQueueToTopic that binds a queue to a specific routing key on a channel.

The main method is the entry point for the RabbitMQConsumer class. It creates a connection to the RabbitMQ server using the connection factory, and creates a channel to communicate with the server. The channel is then used to create and bind a queue to a specific routing key using the bindQueueToTopic method.

The RabbitMQConsumer class is an essential part of the EduCostStat application, as it allows for the consumption of messages from the RabbitMQ server and the processing of those messages to generate query results.

```java
// Declare exchange type as TOPIC
channel.exchangeDeclare(EXCHANGE_NAME, BuiltinExchangeType.TOPIC);

// Create and bind queue for Query One messages
String queueNameQueryOne = channel.queueDeclare("EduCostStat_query_one", true, false, false, null)
        .getQueue();
bindQueueToTopic(channel, queueNameQueryOne, "EduCostStatQueryOne.#");

// Create and bind queue for Query Two messages
String queueNameQueryTwo = channel.queueDeclare("EduCostStat_query_two", true, false, false, null)
        .getQueue();
bindQueueToTopic(channel, queueNameQueryTwo, "EduCostStatQueryTwo.#");

// Create and bind queue for Query Three messages
String queueNameQueryThree = channel.queueDeclare("EduCostStat_query_three", true, false, false, null)
        .getQueue();
bindQueueToTopic(channel, queueNameQueryThree, "EduCostStatQueryThree.#");

// Create and bind queue for Query Four messages
String queueNameQueryFour = channel.queueDeclare("EduCostStat_query_four", true, false, false, null)
        .getQueue();
bindQueueToTopic(channel, queueNameQueryFour, "EduCostStatQueryFour.#");

// Create and bind queue for Query Five messages
String queueNameQueryFive = channel.queueDeclare("EduCostStat_query_five", true, false, false, null)
        .getQueue();
bindQueueToTopic(channel, queueNameQueryFive, "EduCostStatQueryFive.#");

// Print a message indicating that the consumer is waiting for messages
System.out.println(" [*] Waiting for messages. To exit press CTRL+C");
```

Figure 2.17: Establishing and connecting each queue to its corresponding query through bindings

In this part of the code above, we set up a connection to the RabbitMQ server using the connection factory and create a channel to interact with it. We declare the exchange type as TOPIC and create and bind queues for each of the five queries.

For each queue, we specify a routing key pattern that matches the corresponding query. For example, the queue for Query One messages is bound to the routing key pattern ”EduCostStatQueryOne.#”, which means that it will receive all messages that begin with ”EduCostStatQueryOne”.

Once the queues are set up and bound to their respective routing keys, the program prints a message indicating that the consumer is waiting for messages.

```
// Define callback function for handling messages for Query One
DeliverCallback deliverCallbackQueryOne = (consumerTag, delivery) -> {
    try {
        // Parse the message body into a QueryOneRequest object
        QueryOneRequest request = QueryOneRequest.parseFrom(delivery.getBody());
        System.out.println("Received query one request: " + request);

        // Retrieve data from MongoDB
        MongoCollection<Document> collection = MongoDBUtil.getCollection("EduCostStatQueryOne");
        Document query = new Document("query", Arrays.asList(request.getYear(), request.getState(),
                request.getType(), request.getLength(), request.getExpense()));
        System.out.println("Executing query: " + query.toString());
        Document result = collection.find(query).first();

        // If query result exists, generate a response and publish it to the exchange
        // with appropriate routing key
        if (result != null) {
            int expense = result.getInteger("result");
            System.out.println("Query One result: " + expense);

            // Generate the response
            QueryOneResponse response = QueryOneResponse.newBuilder().setQueryId("1")
                    .setTotalExpense(expense).build();

            // Publish the response message to the exchange with the appropriate routing key
            String routingKey = String.format("QueryOneResponse.%d.%s.%s.%s.%s", request.getYear(),
                    request.getState(), request.getType(), request.getLength(), request.getExpense());
            channel.basicPublish(EXCHANGE_NAME, routingKey, null, response.toByteArray());
            System.out.println("Sent query one response: " + response);
        } else {
            System.out.println("Query result not found in the collection.");
        }

    } catch (InvalidProtocolBufferException e) {
        // Handle errors when parsing the message body
        System.out.println("Invalid message received for query one: " + e.getMessage());
    }
};
```

Figure 2.18: The callback function for handling messages related to Query One. When a message related to Query One is received by the consumer, this function is triggered to process and handle the message.

In this code snippet above, a DeliverCallback function is defined for handling messages received by the consumer for Query One. The function is passed as a parameter to the channel’s basicConsume() method which sets up the consumer to receive messages on the queue associated with Query One requests.

Inside the function, the message body is parsed into a QueryOneRequest object and the required data is retrieved from MongoDB using a query. If a result is found, a QueryOneResponse is generated with the required information and published to the exchange with an appropriate routing key. If no result is found, a message indicating that the query result was not found is printed to the console.

The function also includes an error handling code to catch and print error messages related to parsing the message body.

```
Console  Console ×
<terminated> RabbitMQConsumer [Java Application] C:\Program Files\Java\jdk-1
 [*] Waiting for messages. To exit press CTRL+C
Received query one request: year: 2018
state: "Arizona"
type: "Private"
length: "4-year"
expense: "Fees/Tuition"
```

Figure 2.19: Consumer Requesting Message for Query One from Producer

```
Query One result: 13487
Sent query one response: queryId: "1"
totalExpense: 13487
```

Figure 2.20: Consumer receives the message from Producer for Query One

```java
// Define callback function for handling messages for Query Two
DeliverCallback deliverCallbackQueryTwo = (consumerTag, delivery) -> {
    try {
        // Parse the message body into a QueryTwoRequest object
        QueryTwoRequest request = QueryTwoRequest.parseFrom(delivery.getBody());
        System.out.println("Received query two request: " + request);

        // Retrieve data from MongoDB
        List<Document> expensiveStates = QueryTwoDAO.query(request.getYear(), request.getType(),
                request.getLength());

        // Build the response using the list of documents
        QueryTwoResponse.Builder responseBuilder = QueryTwoResponse.newBuilder();
        for (Document doc : expensiveStates) {
            String state = doc.getString("_id");
            int total = doc.getInteger("total");
            ExpensiveState expensiveState = ExpensiveState.newBuilder().setState(state).setTotal(total)
                    .build();
            responseBuilder.addExpensiveStates(expensiveState);
        }
        QueryTwoResponse response = responseBuilder.build();

        // Publish the response message to the exchange with the appropriate routing key
        String routingKey = String.format("QueryTwoResponse.%d.%s.%s", request.getYear(), request.getType(),
                request.getLength());
        channel.basicPublish(EXCHANGE_NAME, routingKey, null, response.toByteArray());
        System.out.println("Sent query two response: " + response);

    } catch (InvalidProtocolBufferException e) {
        // Handle errors when parsing the message body
        System.out.println("Invalid message received for query two: " + e.getMessage());
    }
};
```

Figure 2.21: The callback function for handling messages related to Query Two. When a message related to Query Two is received by the consumer, this function is triggered to process and handle the message.

```
Received query two request: year: 2013
type: "Private"
length: "4-year"
```

Figure 2.22: Consumer Requesting Message for Query Two from Producer

```
Query already exists in the collection.
Top 5 most expensive states:
Massachusetts: 49871.0
District of Columbia: 48440.0
Connecticut: 48262.0
Vermont: 46255.0
Rhode Island: 46114.0
Sent query two response: expensiveStates {
  state: "Massachusetts"
  total: 49871
}
expensiveStates {
  state: "District of Columbia"
  total: 48440
}
expensiveStates {
  state: "Connecticut"
  total: 48262
}
expensiveStates {
  state: "Vermont"
  total: 46255
}
expensiveStates {
  state: "Rhode Island"
  total: 46114
}
```

Figure 2.23: Consumer receives the message from Producer for Query Two

```java
// Define callback function for handling messages for Query Three
DeliverCallback deliverCallbackQueryThree = (consumerTag, delivery) -> {
    try {
        // Parse the message body into a QueryThreeRequest object
        QueryThreeRequest request = QueryThreeRequest.parseFrom(delivery.getBody());
        System.out.println("Received query three request: " + request);

        // Query the database for the economic states
        List<Document> economicStates = QueryThreeDAO.query(request.getYear(), request.getType(),
                request.getLength());

        // Build the response using the list of documents
        QueryThreeResponse.Builder responseBuilder = QueryThreeResponse.newBuilder();
        for (Document doc : economicStates) {
            String state = doc.getString("_id");
            int total = doc.getInteger("total");
            EconomicState economicState = EconomicState.newBuilder().setState(state).setTotal(total)
                    .build();
            responseBuilder.addEconomicStates(economicState);
        }
        QueryThreeResponse response = responseBuilder.build();

        // Publish the response message to the exchange with the appropriate routing key
        String routingKeyResponseQueryThree = String.format("QueryThreeResponse.%d.%s.%s",
                request.getYear(), request.getType(), request.getLength());
        channel.basicPublish(EXCHANGE_NAME, routingKeyResponseQueryThree, null, response.toByteArray());
        System.out.println("Sent query three response: " + response);

    } catch (InvalidProtocolBufferException e) {
        // Handle errors when parsing the message body
        System.out.println("Invalid message received for query three: " + e.getMessage());
    }
};
```

Figure 2.24: The callback function for handling messages related to Query Three. When a message related to Query Three is received by the consumer, this function is triggered to process and handle the message.

```
Received query three request: year: 2021
type: "Public Out-of-State"
length: "4-year"
```

Figure 2.25: Consumer Requesting Message for Query Three from Producer

```
Query already exists in the collection.
Sent query three response: economicStates {
  state: "District of Columbia"
  total: 13004
}
economicStates {
  state: "South Dakota"
  total: 21090
}
economicStates {
  state: "North Dakota"
  total: 22493
}
economicStates {
  state: "Wyoming"
  total: 24509
}
economicStates {
  state: "Florida"
  total: 29325
}
```

Figure 2.26: Consumer receives the message from Producer for Query Three

```java
// Define callback function for handling messages for Query Four
DeliverCallback deliverCallbackQueryFour = (consumerTag, delivery) -> {
    try {
        // Parse the message body into a QueryFourRequest object
        QueryFourRequest request = QueryFourRequest.parseFrom(delivery.getBody());
        System.out.println("Received query four request: " + request);

        // Query the database for the growth rates
        List<Document> growthRates = QueryFourDAO.query(request.getBaseYear(), request.getYearsAgo(),
                request.getType(), request.getLength());

        // Build the response using the list of documents
        QueryFourResponse.Builder responseBuilder = QueryFourResponse.newBuilder();
        for (Document doc : growthRates) {
            String state = doc.getString("state");
            double growthRate = doc.getDouble("growthRate");
            StateGrowthRate growthRateObj = StateGrowthRate.newBuilder().setState(state)
                    .setGrowthRate(growthRate).build();
            responseBuilder.addStateGrowthRates(growthRateObj);
        }
        QueryFourResponse response = responseBuilder.build();

        // Publish the response message to the exchange with the appropriate routing key
        String routingKeyResponseQueryFour = String.format("QueryFourResponse.%d.%d.%s.%s",
                request.getBaseYear(), request.getYearsAgo(), request.getType(), request.getLength());
        channel.basicPublish(EXCHANGE_NAME, routingKeyResponseQueryFour, null, response.toByteArray());
        System.out.println("Sent query four response: " + response);

    } catch (InvalidProtocolBufferException e) {
        // Handle errors when parsing the message body
        System.out.println("Invalid message received query four: " + e.getMessage());
    }
};
```

Figure 2.27: The callback function for handling messages related to Query Four. When a message related to Query Four is received by the consumer, this function is triggered to process and handle the message.

```
Received query four request: baseYear: 2014
yearsAgo: 3
type: "Private"
length: "4-year"
```

Figure 2.28: Consumer Requesting Message for Query Four from Producer

```
Query result already exists in the collection.
Idaho: 7.5623700623700625
Wyoming: 7.516098903799833
Iowa: 3.3845508730943186
New Mexico: 3.079472393451316
South Dakota: 2.232904326252132
```

Figure 2.29: Consumer receives the message from Producer for Query Four

```java
// Define callback function for handling messages for Query Five
DeliverCallback deliverCallbackQueryFive = (consumerTag, delivery) -> {
    try {
        // Parse the message body into a QueryFiveRequest object
        QueryFiveRequest request = QueryFiveRequest.parseFrom(delivery.getBody());
        System.out.println("Received query five request: " + request);

        // Query the database for region costs
        Map<String, Double> regionCostMap = QueryFiveDAO.query(request.getYear(), request.getType(),
                request.getLength());

        // Build the response using the map of region costs
        QueryFiveResponse.Builder responseBuilder = QueryFiveResponse.newBuilder();
        for (Map.Entry<String, Double> entry : regionCostMap.entrySet()) {
            RegionCost.Builder regionCostBuilder = RegionCost.newBuilder().setRegion(entry.getKey());
            Double cost = entry.getValue();
            if (cost != null) {
                regionCostBuilder.setCost(cost);
            }
            responseBuilder.addRegionCosts(regionCostBuilder.build());
        }
        QueryFiveResponse response = responseBuilder.build();

        // Publish the response message to the exchange with the appropriate routing key
        String routingKeyResponseQueryFive = String.format("QueryFiveResponse.%d.%s.%s", request.getYear(),
                request.getType(), request.getLength());
        channel.basicPublish(EXCHANGE_NAME, routingKeyResponseQueryFive, null, response.toByteArray());
        System.out.println("Sent query five response: " + response);

    } catch (InvalidProtocolBufferException e) {
        // Handle errors when parsing the message body
        System.out.println("Invalid message received for query five: " + e.getMessage());
    }
};
```

Figure 2.30: The callback function for handling messages related to Query Five. When a message related to Query Five is received by the consumer, this function is triggered to process and handle the message.

```
Received query five request: year: 2018
type: "Private"
length: "4-year"
```

Figure 2.31: Consumer Requesting Message for Query Five from Producer

```
Query result already exists in the collection.
Region-wise average overall expense for year: 2018, type: Private, and length: 4-year
Midwest: 18553.458333333332
SouthEast: 16811.25
West: 16631.5
SouthWest: 19961.0
NorthEast: 25828.958333333332
```

Figure 2.32: Consumer receives the message from Producer for Query Five

```java
// Define the cancelCallback to be executed when the consumer is canceled
CancelCallback cancelCallback = consumerTag -> {
    // Decrement the latch to signal that the consumer has been canceled
    latch.countDown();
};

// Start consuming messages from each queue, using the appropriate callback
// function
channel.basicConsume(queueNameQueryOne, true, deliverCallbackQueryOne, cancelCallback);
channel.basicConsume(queueNameQueryTwo, true, deliverCallbackQueryTwo, cancelCallback);
channel.basicConsume(queueNameQueryThree, true, deliverCallbackQueryThree, consumerTag -> {
});
channel.basicConsume(queueNameQueryFour, true, deliverCallbackQueryFour, consumerTag -> {
});
channel.basicConsume(queueNameQueryFive, true, deliverCallbackQueryFive, consumerTag -> {
});

// Wait indefinitely until the consumer is canceled
latch.await();
```

Figure 2.33: The message callback function is triggered whenever a new message is available in the queue. The message is then processed by the callback function according to the query that the queue is responsible for handling.

3 Conclusion

In conclusion, this project successfully demonstrates the implementation of a RabbitMQ-based system to handle data retrieval and processing tasks. The producer and consumer applications have been designed to interact with a MongoDB cloud service, retrieve datasets based on topics, and customize the data processing using a configuration file. The producer application publishes the data to the appropriate exchange topics with matching routing keys, while the consumer application subscribes to the specific topics of interest and processes the received data accordingly.

By completing Task 1, we have set up the RabbitMQ server on the local computer, providing a robust message broker infrastructure to facilitate communication between the producer and consumer applications.

In Task 2, we programmed the producer and consumer using RabbitMQ exchange topic libraries. This setup ensures that the producer and consumer run independently, without relying on multi-threading within the same application.

The project showcases a scalable and flexible approach to handling data retrieval and processing tasks. The use of RabbitMQ ensures reliable message delivery and supports the decoupling of the producer and consumer applications, which allows for independent scaling and improves fault tolerance. Additionally, the configuration file-driven approach offers flexibility in customizing data retrieval and processing without altering the codebase.

In summary, this project demonstrates the effective use of RabbitMQ and MongoDB cloud services to create a robust, scalable, and flexible system for data retrieval and processing. The design and implementation of the producer and consumer applications, along with the successful completion of the tasks, have achieved the project objectives and provided valuable insights into building efficient and practical data processing systems.