Assignments Covered

- Working with AWS Databases and Lambda: Postgres (Base Task)
- Working with AWS Databases and Lambda: Postgres (Bonus 3)
- Working with AWS Databases and the Command Line/Terminal: DynamoDB (Base Task)

(The tutorial was also finished, but that was simply following the instructions provided)

All work was done in region us-west-2(Oregon) whenever possible.

###

Working with AWS Databases and Lambda: Postgres (Base Task)

I created the RDS database by following the tutorial provided and called it “rds-postgres-tutorial” with the master username “postgres” and the password “postgres.”

Afterwards, I set up two lambda functions: one to stop the server, and one to start it. The process for creating both is somewhat similar. I will go over the process for setting up the lambda function to stop the server and cover the differences between it and the function to start the server.

I created an IAM policy called “RDS_Start_Stop_Policy” that gave permissions to check for existing databases on my account as well as permissions to start/stop database instances using the following JSON:

```
{
    "Version": "2012-10-17",
       "Statement": [
       {
              "Effect": "Allow",
              "Action": [
              "rds:DescribeDBInstances",
              "rds:StopDBInstance",
              "rds:StartDBInstance"
              ],
              "Resource": "*"
       },
       {
              "Effect": "Allow",
        "Action": [
              "logs:CreateLogGroup",
              "logs:CreateLogStream",
              "logs:PutLogEvents"
              ],
```

```
            "Resource": "*"
        }
        ]
}
```

After that, I created the role “AWSLambdaRole” in the IAM Management Console. This role can be used with Lambda functions and utilizes “RDS_Start_Stop_Policy.”

==(From here, the function to start the database undergoes a similar, but separate process)==

Next, I created the Lambda function “RDSInstanceStop.” This function utilizes Python 3.8 and uses the IAM Role “AWSLambdaRole” for its permissions. Using the ARN of the function after creation, I also created a new inline policy called “policy_rds_stop” using the following JSON code:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:GetFunctionConfiguration",
            "Resource":
"arn:aws:lambda:us-west-2:367829174649:function:RDSInstanceStop"
        }
    ]
}
```

The string following “Resource” is the ARN of “RDSInstanceStop.”

After that, I turned towards the code of the Lambda function itself. I used the following Python 3.8 code:

```
import sys
import botocore
import boto3
from botocore.exceptions import ClientError

def lambda_handler(event, context):
    rds = boto3.client('rds')
    lambdaFunc = boto3.client('lambda')
    print ('Trying to get Environment variable')
    try:
        funcResponse = lambdaFunc.get_function_configuration(
            FunctionName='RDSInstanceStop'
        )
        DBinstance = funcResponse['Environment']['Variables']['DBInstanceName']
        print ('Stopping RDS service for DBInstance : ')
    except ClientError as e:
        print(e)
    try:
        response = rds.stop_db_instance(
```

```
            DBInstanceIdentifier=DBinstance
        )
        print ('Success')
    except ClientError as e:
        print(e)

    return {
        'message' : "Script execution completed. See Cloudwatch logs for complete
output"
    }
```

After deploying this code, I created a default test and ran said test. The result is that the server enters a “Stopping” status for a few minutes before finally stopping.

To create the Lambda function that starts the server back up again, I used a similar process to create the function “RDSInstanceStart.” Slight changes of the JSON and Python 3.8 code can be found in the zipped folder.

###

Working with AWS Databases and Lambda: Postgres (Bonus 3)

For this task, I recreated an instance of the database “rds-postgres-tutorial” in CloudFormation by using the CloudFormation console.

First, I selected to create a new stack using existing resources. I was prompted to provide a template for this stack, which is shown below:

```
{
    "Resources" : {
        "MyDB" : {
            "Type" : "AWS::RDS::DBInstance",
            "Properties" : {
                "AllocatedStorage" : "20",
                "DBInstanceClass" : "db.t2.micro",
                "Engine" : "PostgreSQL",
                "MasterUsername" : "postgres",
                "MasterUserPassword" : "postgres"
            },
            "DeletionPolicy" : "Snapshot"
        }
    }
}
```

The information in “Properties” was gathered from the overview of “rds-postgres-tutorial” in the RDS console.

The other major thing I did for setup is referring the stack to “rds-postgres-tutorial” for duplication. Upon finishing the setup, the stack began importing and updating the information.

Looking back, one way I could have improved this replication was to include the security group used in “rds-postgres-tutorial.” Since I didn’t declare what security group I was using, it replaced the new security group I set up using the tutorial to a default security group. This specific update took a somewhat long time (about 20 minutes) and I believe I could have avoided this if I specified the security group in the template.

###

Working with AWS Databases and the Command Line/Terminal: DynamoDB (Base Task)

After installing AWS CLI, I went on to the IAM Management Console and added a user “ayhu0311” with admin privileges, Access Key, and Secret Access Key. From there, I configured that user using **aws configure** in the terminal.

Once I’ve finished setting up the user “ayhu0311” as the base user on the command line, I created a new table in DynamoDB using the following command line input:

```
aws dynamodb create-table \
--table-name Shops \
--attribute-definitions \
AttributeName=ShopName,AttributeType=S \
--key-schema \
AttributeName=ShopName,KeyType=HASH \
--provisioned-throughput \
ReadCapacityUnits=10,WriteCapacityUnits=5
```

This creates the table “Shops” with the primary key attribute “ShopName” that utilizes a hash index.

After creating the table, I inserted an item using the following command line input:

```
aws dynamodb put-item \
--table-name Shops \
--item '{
"ShopName": {"S": "Shop1"},
"Time": {"N": "15"},
"Price": {"N": "15.00"}
}'
```

This inserts a row with the attributes “ShopName,” “Time,” and “Price.”

(The practical use for this table would be to insert a bunch of different shops that all sell some one particular item/service of interest at various prices and have different travel times to reach.)

Once I've inserted an item, I retrieved it using the following command line input:

```
aws dynamodb get-item \
--table-name Shops --key '{"ShopName": {"S":"Shop1"}}'
```

###

References

Besides official documentation, I also referred to the following link to set up the lambda functions for starting/stopping the database:

https://www.sqlshack.com/automatically-start-stop-an-aws-rds-sql-server-using-aws-lambda-functions/