**TCSS 562: Software Engineering for Cloud Computing**
Fall 2021
http://faculty.washington.edu/wlloyd/courses/tcss562

School of Engineering and Technology
University of Washington – Tacoma

# Tutorial 6 – Introduction to Lambda III: Serverless Databases

*Disclaimer: Subject to updates as corrections are found*
Version 0.11
Scoring: 20 pts maximum

The purpose of this tutorial is to introduce the use of relational databases from AWS Lambda. This tutorial will demonstrate the use of SQLite, an in-memory file-based database that runs inside a Lambda function to provide a "temporary" relational database that lives for the lifetime of the Lambda container. Secondly, the tutorial demonstrates the use of the Amazon Relational Database Service (RDS) to create a persistent relational database using Serverless Aurora MySQL 5.6 for data storage and query support for Lambda functions.

Goals of this tutorial include:

1. Introduce the Sqlite database using the command line "sqlite3" tool.
2. Deploy a Lambda Function that uses a file-based SQLite3 database in the "/tmp" directory of the Lambda container that persists between client function invocations
3. Compare the difference between using file-based and in-memory SQLite DBs on Lambda.
4. Create an Amazon RDS Aurora MySQL Serverless database
5. Launch an EC2 instance and install the mysql command line client to interface with the Aurora serverless database.
6. Deploy an AWS Lambda function that uses the MySQL Serverless database.

## 1. Using the SQLite Command Line

To begin, create a directory called "saaf_sqlite".
Then clone the git repository under the new directory:

```
git clone https://github.com/wlloyduw/saaf_sqlite3.git
```

If using Windows or Mac, download the "Precompiled binaries" as a zip file from:
https://www.sqlite.org/download.html

On Windows/Mac, unzip the zip file, and then run the **sqlite3** program.

On Ubuntu Linux, the package sqlite3 can be installed which is version ~3.31.1 on Ubuntu 20.04 LTS. Then launch the sqlite3 database client:

```
sudo apt update
sudo apt install sqlite3
# navigate to your java project directory first
cd {base directory where project was cloned}/saaf_sqlite3/java_template/
sqlite3
```

Check out the version of the db using ".version".
Check out available commands using ".help".

```
sqlite> .version
SQLite 3.31.1 2020-01-27 19:55:54
zlib version 1.2.11
gcc-9.3.0
sqlite> .quit
```

Start by saving a new database file, and then exit the tool:

```
sqlite> .save new.db
sqlite> .quit
```

Then, check the size of an empty sqlite db file:

```
$ ls -l new.db
total 3848
-rw-r--r-- 1 wlloyd wlloyd    4096 Nov  1 19:05 new.db
```

It is only 4096 bytes, very small!

Next, work with data in the database:

```
$ sqlite3 new.db
SQLite version 3.31.1 2020-01-27 19:55:54
Enter ".help" for usage hints.
sqlite> .databases
main: /home/wlloyd/git/tutorial6/saaf_sqlite3/new.db
sqlite> .tables
```

There are initially no tables.
Create a table and insert some data:

```
sqlite> create table newtable (name text, city text, state text);
sqlite> .tables
newtable
sqlite> insert into newtable values('Susan Smith','Tacoma','Washington');
sqlite> insert into newtable values('Bill Gates','Redmond','Washington');
sqlite> select * from newtable;
Susan Smith|Tacoma|Washington
Bill Gates|Redmond|Washington
```

Now check how the database file has grown after adding a table and a few rows:

sqlite> .quit
$ ls -l new.db

**Question 1. After creating the table 'newtable' and loading data to sqlite, what is the size of the new.db database file?**

The sqlite3 command line tool can be used to perform common **C**reate **R**ead **U**pdate and **D**elete queries on a sqlite database. This allows the database to be preloaded with data and bundled with a Lambda function for deployment to the cloud as needed.

If you're unfamiliar with SQL, and writing SQL queries, please consider completing the online tutorial:

SQLite Tutorial:
http://www.sqlitetutorial.net/

Follow the steps for "Getting started with SQLite" (1, 2, and 3), and then complete the Basic SQLite tutorial to review performing different types of queries using the sample Chinook database with 11 tables downloaded from step 3.

**2. Combining SQLite with AWS Lambda**

SQLite can be leveraged directly from programming languages such as Java, Python, and Node.JS. SQLite provides an alternative to basic CSV and text file storage with a SQL-compatible query-able file format. SQLite does not replace a full-fledged enterprise relational database management system (dbms) in terms of scalability, etc. But given the small footprint of SQLite, it provides an excellent database alternative for serverless environments and Internet of Things devices.

Next, explore the saaf_sqlite project in Netbeans or another Java IDE.

"saaf_sqlite" provides a Java-based Lambda "Hello" function based on SAAF from Tutorial #4. Look at the code inside: **saaf_sqlite/java_template/src/main/java/ lambda/HelloSqlite.java**.

```
setCurrentDirectory("/tmp");
try
{
      // Connection string an in-memory SQLite DB
      //Connection con = DriverManager.getConnection("jdbc:sqlite:");

      // Connection string for a file-based SQlite DB
      Connection con = DriverManager.getConnection("jdbc:sqlite:mytest.db");
```

The first line of code (LOC) calls a helper function to set the working directory to "/tmp" inside the Lambda function.

"/tmp" provides a read/write 512MB filesystem on Lambda.
As a security precaution, code deployed to Lambda has only limited permission to write to the filesystem, as /tmp is enabled for read/write.

SQLite can work with databases entirely in memory, or on disk. The first database connection string is commented out, but could be used if wanting to work with a database only in memory.

The advantage of creating the database on disk is that data persists beyond the runtime of the Java code. On Lambda, this means as long as the original runtime container is preserved, the data is preserved. If containers

are kept WARM, they can last up to 6 to 8 hours. After 6-8 hours, it will be necessary to save any SQLite databases to S3 to persist the data for longer.

Perform a clean build of the saaf_sqlite project to create a jar file.
Following instructions from tutorial #4, deploy a new lambda function called "helloSqlite".
Be sure to set the function's handler in the AWS Lambda GUI.
Choose one method (AWS CLI or CURL) for invoking "helloSqlite" from **callservice.sh**.
If wanting to use a HTTP/REST URL, configure the API Gateway to provide a URL for access via Curl. Otherwise use the "helloSqlite" Lambda function name and the AWS Lambda CLI.

Under your new project, modify the callservice.sh script to invoke your newly deployed Sqlite Lambda function:
**saaf_sqlite/java_template/test/callservice.sh**

Then run the script. Below the API Gateway invocation code in BASH has been commented out using a "#" in front of each line.

```
$ ./callservice.sh
Invoking Lambda function using AWS CLI
real 0m11.985s
user 0m0.288s
sys 0m0.064s

AWS CLI RESULT:
{
  ….. // some attributes removed from brevity...
  "uuid": "8c321d18-d16e-4cd8-acac-cbc8d65fe138",
  "error": "",
  "vmuptime": 1541129227,
  "newcontainer": 1,
  "value": "Hello Fred Smith",
  "names": [
    "Fred Smith"
  ]
}
```

Using a file, each time the service is called and the same runtime container is used, a name is appended to the temporary file-based SQLite database. We see the "names" array in the JSON grow with each subsequent call.

Try running the ./callservice.sh script now several times (10x) to watch the names array grow.

Now, try out what happens when two clients call the Lambda function at the same time.

Inspect the simple calltwice.sh script:

```
cat calltwice.sh
```

Now, try running calltwice.sh:

```
./calltwice.sh
```

If you do not see the command prompt after awhile, press [ENTER].

Invoking a Lambda with two clients in parallel forces Lambda to create additional server infrastructure.

**Question 2. When the second client calls the helloSqlite Lambda function, how is the data different in the second container environment compared to the initial/first container?**

Now, try out a memory-only SQLite database. Modify your Lambda code to swap out the type of database. Comment out the file-based database in favor of memory only:

```
setCurrentDirectory("/tmp");
try
{

  // Connection string an in-memory SQLite DB
  Connection con = DriverManager.getConnection("jdbc:sqlite:");

  // Connection string for a file-based SQlite DB
  // Connection con = DriverManager.getConnection("jdbc:sqlite:/tmp/mytest.db");
```

Build a new JAR file, and redeploy it to Lambda for the helloSqlite Lambda function.

Using callservice.sh, try calling the Lambda several times in succession.

**Question 3. For Lambda calls that execute in the same runtime container identified by the UUID returned in JSON, does the data persist between client Lambda calls with an in-memory DB? (YES or NO)**

Next, let's modify the code for helloSqlite to add a static int counter that tracks the total number of calls to the container.

Define a static int at the start of public class HelloSqlite:

```
public class HelloSqlite implements RequestHandler<Request, HashMap<String,
Object>>
{
    static String CONTAINER_ID = "/tmp/container-id";
    static Charset CHARSET = Charset.forName("US-ASCII");

   static int uses = 0;
```

Then modify the definition of String hello near the bottom of the Lambda function to report the uses count:

```
        // ***************************************************************
        // Set hello message here
        // ***************************************************************
      uses = uses + 1;
        String hello = "Hello " + request.getName() + " calls=" + uses;
```

Build a new JAR file, and redeploy it to Lambda for the helloSqlite Lambda function.

Using callservice.sh, try calling the Lambda with the static uses counter several times in succession:

```
./callservice.sh
./callservice.sh
./callservice.sh
```

**Question 4. Does the value of the static int persist for Lambda calls that execute in the same runtime container identified by the UUID returned in JSON? (YES or NO)**

Now, try running with calltwice.sh.

**Question 5. How is the value of the static int different across different runtime containers identified by the UUID returned in JSON?**

Next, inspect the SQL code for the helloSqlite Lambda function:

```
// Detect if the table 'mytable' exists in the database
PreparedStatement ps = con.prepareStatement("SELECT name FROM sqlite_master WHERE
type='table' AND name='mytable'");
ResultSet rs = ps.executeQuery();
if (!rs.next())
{
      // 'mytable' does not exist, and should be created
      logger.log("trying to create table 'mytable'");
      ps = con.prepareStatement("CREATE TABLE mytable ( name text, col2 text, col3
text);");
      ps.execute();
}
rs.close();

// Insert row into mytable
ps = con.prepareStatement("insert into mytable values('" + request.getName() +
"','b','c');");
ps.execute();

// Query mytable to obtain full resultset
ps = con.prepareStatement("select * from mytable;");
rs = ps.executeQuery();
```

The approach of our helloSqlite Lambda is to create a new file (or memory) database each time.

**Question 6. Before inserting rows into 'mytable', what has to be done and why in the Java code above?**

Consider how an in-memory SQLite DB could be preserved between calls. The Lambda function could use a static Connection object that is initialized similar to how the "uses" variable is initialized above. The connection would stay open as long as the Lambda function remains "warm".

**3. Optional Exercise: Persisting SQLite database files to S3**

Leveraging concepts from tutorial #5, modify the file-based version of helloSqlite to always save the database file in /tmp to S3 at the end of the function. Add a key to the request.java to allow the user to specify a database filename.

At the beginning of the function handler, using the user provided database filename from the request, check if the specified file exists in /tmp. If it does not exist, then download the file from S3. Then change the SQLite connection string to open the user provided database name. This way a user could request a specific database from S3 for their Lambda function call. As an added feature, if the file does not exist in S3, then create an initial version in /tmp and upload this to S3.

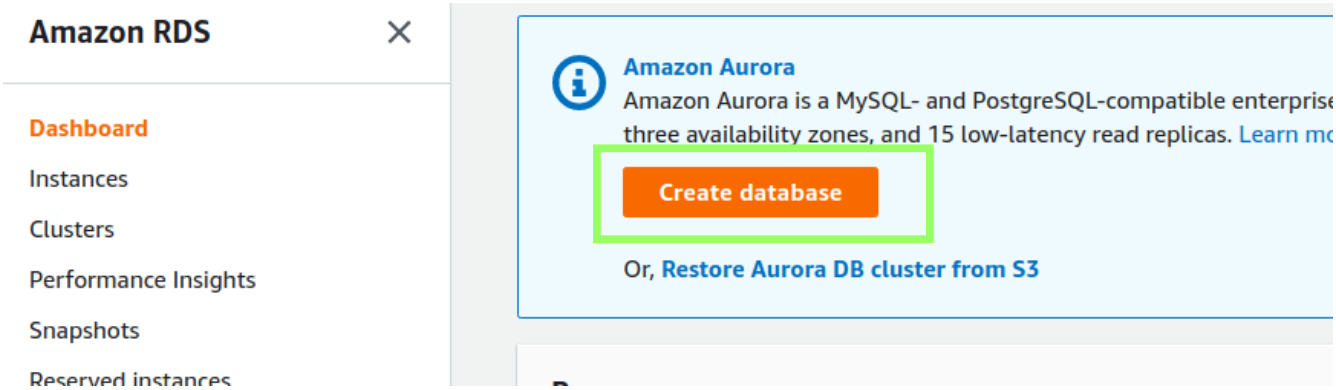
**4. Create a AWS RDS Aurora MySQL Serverless Database**

The AWS Relational Database Service now offers a serverless MySQL database. The advantage of the serverless database is that it automatically goes idle when not actively used. When idle, the only charges are for storage. Serverless Aurora supports automatic horizontal scaling of database servers up to ~15 nodes. The nodes provide read replicas, where a master is used for database writes. (*Fall 2018: Multiple R/W masters, called "multi-master" is presently in beta*) Vertical scaling allows the CPU and memory resources for the master R/W node to scale from 2 vCPUs and 4GB RAM presumably up to 64 vCPUs and 488 GB RAM, aligning with r4 instance types: **https://www.ec2instances.info/?filter=r4** Aurora additionally provides automatic database backups and replicas.

To get started, lets create a serverless database!

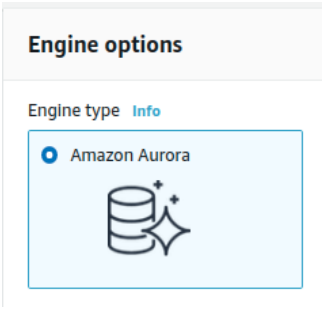
**** This portion of the tutorial requires AWS credits to complete. ****
**Aurora serverless does not run in the FREE tier.**

Under Services, search for and go to "RDS". Click "Create database" to launch the wizard:

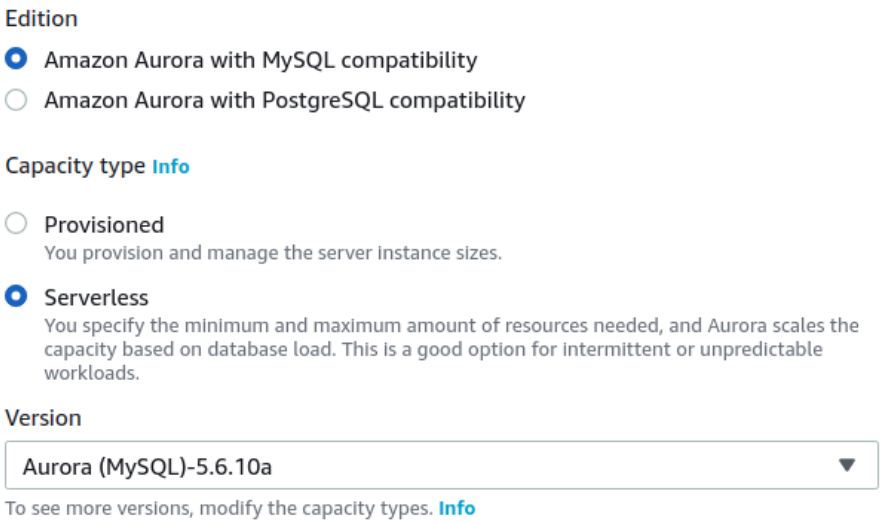



First use the "**Standard create"** database creation method.

Then specify the "Amazon Aurora" engine

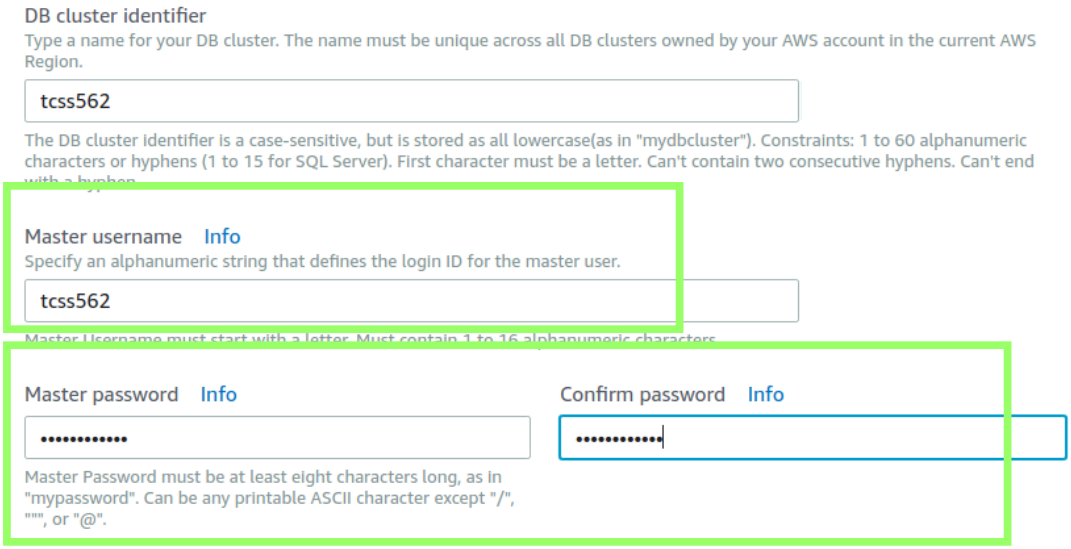



Then choose the following options:
Be sure to select “Serverless”:

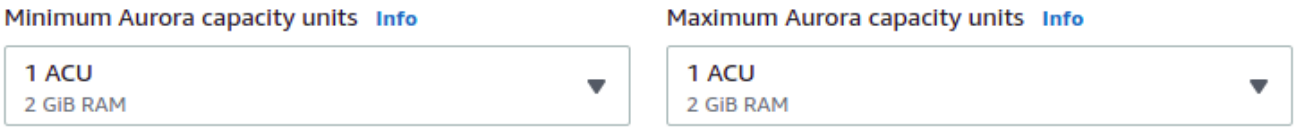



Aurora MySQL-5.6.10a is the only “serverless” MySQL database version supported.

Next, values for the ‘DB cluster identifier’, ‘Master username’, as well as the password.
Be sure to set and remember the password:

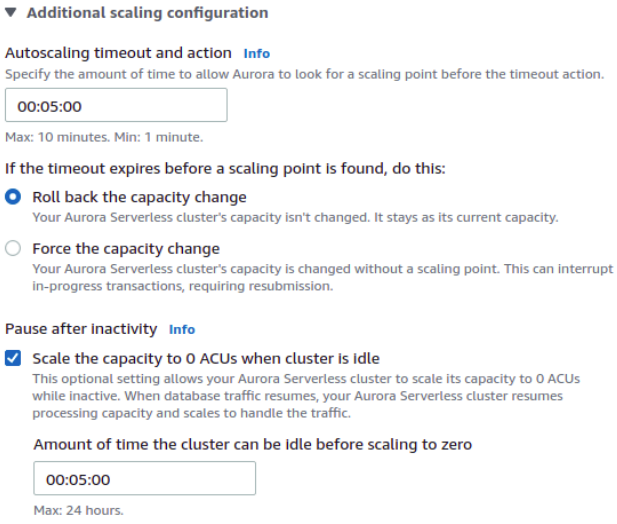



Next, provide scaling specifications. To save cost, specify the minimum possible scaling:




Note that running the server for 1 hour with 1 Aurora Capacity Unit and 2GB of memory costs 6¢. Aurora Serverless costs 6¢/per ACU/per hour billed to the nearest second with a 5-minute minimum each time the database is activated. To limit the maximum per hour charge, set the maximum Aurora capacity units to the minimum setting **1**. This effectively disables auto-scaling.

Open the "Additional scaling configuration":




**** ENABLE THE OPTION FOR "SCALE THE CAPACITY TO 0 ACUs WHEN CLUSTER IS IDLE"… THIS IS CRUCIAL OTHERWISE THE DATABASE WILL HAVE A CONTINUOUS CHARGE… !!!**

One Aurora Capacity Unit (ACU) has approximately 2 GB of memory with corresponding CPU and networking, similar to that of Aurora Standard instances "r4" instances (see above link).

For Connectivity specify: Virtual Private Cloud: Default VPC

Then open "Additional connectivity configuration":

Subnet group: Select an existing DB subnet group if available, otherwise select: "Create new DB Subnet Group"
VPC security groups: Choose existing, then select "default" from the dropdown list.

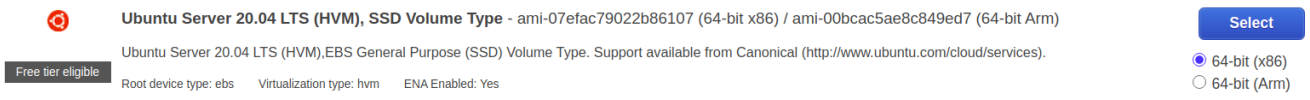
For the remaining settings, the defaults can be used.
The press:


The RDS databases list will appear.
The state of creation can be monitored. Database creation takes a couple minutes.


**5. Launch a t2.micro EC2 VM to connect to the Aurora DB MySQL database**

It is not possible to directly connect to the Aurora MySQL Serverless database. This is because the database lives on a Virtual Private Cloud (VPC) that does not allow inbound traffic from the internet. This provides network isolation and security. Accessing the database requires launching an EC2 instance in the same VPC as the RDS database and associating a Public IP address with this EC2 instance. The RDS database itself does not have a public IP that is accessible from the outside. Providing direct connectivity to the RDS database requires: (1) setting up either a NAT Gateway (4.5¢/hour), (2) configuring a VM to act as a router/gateway which requires special network configuration, or (3) installing a proxy server such as haproxy on the publicly accessible VM to proxy inbound traffic for MySQL to the RDS database. A database client could then connect to the VM, not RDS, and the traffic is redirected. Fortunately, if deploying AWS Lambda functions in the same VPC as the RDS database, no special networking appliance (e.g. NAT gateway instance or router) is required to access the database. This saves cost and complexity.

Navigate to EC2.
Click on "Launch Instance"
Select the latest version of Ubuntu:




Click [Select].
Select a t2.micro, a free-tier instance. New accounts have 750 hours of free t2.micro time/month.
Click [Next: Configure Instance Details]
Provide the following settings:

Network: vpc (default)  - select your default VPC -
Subnet: no preference
Auto-assign Public IP: Enabled

Click [Next: Add Storage], then
Click [Next: Add Tags], then
Click [Next: Configure Security Group], then

Choose “Select an existing security group”.
And check the “default”.

Then click “Review and Launch”.

Choose an existing keypair from a prior tutorial if available.
Otherwise, create a new keypair.

In the EC2 console, select the VM.  If you have not enabled SSH access from your network, select the new VM, and click on “default” for Security groups.  Click the “Inbound” Tab, and hit the [Edit] button.  Click [Add Rule] and add a “SSH” “TCP” “22” rule for “My IP”.  This should allow SSH access to the t2.micro instance.


Next in the EC2 console copy the public IP:
The right-hand COPY icon makes it easy to copy the IP address.


Now, using the command line, navigate to the folder where the keypair is stored, and ssh into the newly created t2.micro VM.  Paste the address and SSH:

```
$ ssh -i <your key file name> ubuntu@<t2.micro IPv4 public IP>
```

Now on the Ubuntu t2.micro instance launched in the same VPC as the RDS database run the commands:

```
sudo apt update
sudo apt upgrade
```

Optionally, install the AWSCLI if wanting to work with AWS directly from the VM.  It is not required.

```
# installing the AWSCLI is optional
apt install awscli
sudo apt install awscli
aws configure
# provide ACCESS_KEY and SECRET_KEY
# Find your credentials on your existing VM with: "cat ~/.aws/credentials"
```

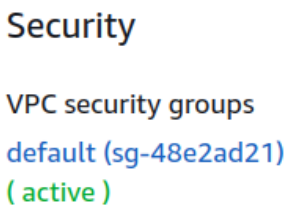
Next, install the mysql client to support connecting to the new RDS database:

```
# Install mysql client
# sudo apt install mysql-client-core-5.7   # For old-versions of Ubuntu < 20.04
sudo apt install mysql-client-core-8.0     # for Ubuntu >= 20.04
```

Now customize the following command to point at your RDS database.

Navigate back into RDS in the AWS management console.
On the left hand-side select “Databases”, then select “tcss562”.

It is necessary to configure the security group to allow the t2.micro to connect to the database. In the Connectivity & security tab, look under “Security” on the right:

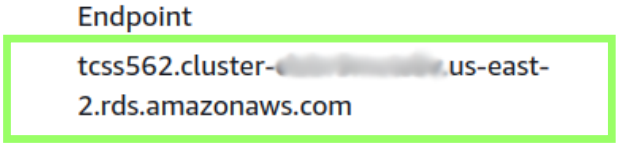



Click on the blue security group label to jump to the EC2 dashboard to edit network security settings.

In the Security editor, click the [Inbound] tab.
Click [Edit].
Click [Add Rule].
Select “MYSQL/Aurora”.
For the source, select “Anywhere” to obtain the address range of “0.0.0.0/0”.
This enables any VM within the private VPC network to be able to connect to the database.
Hit [Save].

Now navigate back to the **RDS service**.
On the left hand-side select “Databases”, then select the “tcss562” DB.
Scroll down and look under “Connectivity and security”.
Copy and paste the name of the “Endpoint”.

For example, copy the Endpoint name:




Now customize the mysql command to connect to your RDS database.
Replace <Database endpoint> and <your database password>.

```
mysql --host=<Database endpoint> --port=3306 --enable-cleartext-plugin --
user=tcss562 --password=<your database password>
```

The database endpoint can be found by browsing the Aurora database in the Amazon RDS GUI. Under the "Connectivity & security" tab, copy and paste the "Endpoint" name and use this in place of '<Database endpoint>' above.

**IMPORTANT NOTE:** When first connecting to your Amazon RDS connection, if reaching this step in the tutorial took longer than 5-minutes from when the database was initially created, then the RDS database has gone to sleep. In this case, calling "mysql" will take up to 30 seconds to respond. Please wait. It is working. If more than a couple minutes transpires, stop with CTRL-C, and troubleshoot.

After sometime, mysql should connect:

```
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 10
Server version: 5.6.10 MySQL Community Server (GPL)
Copyright (c) 2000, 2018, Oracle and/or its affiliates. All rights reserved.
Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
mysql>
```

Try out the following commands.
MySQL can support multiple databases within a single server.
Display the databases on your RDS database:

```
show databases;
```

Now, create a new database:

```
create database TEST;
```

And check the list again:

```
show databases;
```

It is necessary to issue a "use" command for mysql to direct SQL queries to the database:

```
use TEST;
```

Next, create "mytable" to store data:

```
CREATE TABLE mytable (name VARCHAR(40), col2 VARCHAR(40), col3 VARCHAR(40));
```

Then display the list of known tables in the database:

```
show tables;
```

And describe the structure of the table:

```
describe mytable;
```

Now, try adding some data:

```
insert into mytable values ('fred','testcol2','testcol3');
```

And then check if it was inserted:

```
select * from mytable;
```

Help is available with the "help" command:

```
help
```

Exit mysql with:

```
\q
```

It may be useful to "stop" and "start" your t2.micro ec2 instance that has access to the Amazon RDS database to support working with mysql. If no longer planning to use the ec2instance, **terminate it completely**. Note that "stopped" instances incur storage charges. New AWS accounts receive 30GB of EBS GP2 volume disk space for 1 year for free. After 1 year, the charge is 10¢/GB/month. The Ubuntu t2.small requires 8GB of storage. The annual storage cost after the free introductory year goes to $9.60/year for a stopped instance with 1 - 8GB EBS volume. After 1 year, the t2.micro instance is no longer FREE, but the cost rises to 1.16¢/hour, ~27.8¢/day, ~$8.35/month, or ~$101.62/year for GP2, and slightly less for GP3.

**6. Accessing Aurora Serverless Database from AWS Lambda**

Next, on your development computer, create a directory called "saaf_rds".
Then under the new directory, clone the git repository:

```
git clone https://github.com/wlloyduw/saaf_rds_serverless.git
```

This project, provides a Lambda function that will interact with your Amazon RDS database. It requires "mytable" to have been created under the "TEST" database.
Optionally, it should be possible to create the database and table programmatically from Java if necessary.

Once acquiring the project files, it is necessary to create a file called "db.properties".
There is a template provided. Copy this template to be named "db.properties" and edit this file to specify how to connect to your RDS database:

Find and edit this file:

```
cd saaf_rds_serverless/java_template/src/main/resources/
```

```
cp db.properties.empty db.properties
gedit db.properties
```

The **URL** should be specified as follows:

```
jdbc:mysql://<your database endpoint>:3306/TEST
```

Replace "<your database endpoint>" with the RDS database endpoint used to connect with mysql above. Be sure to add values for **password**, and **username** as well based on how your RDS database was initially configured.

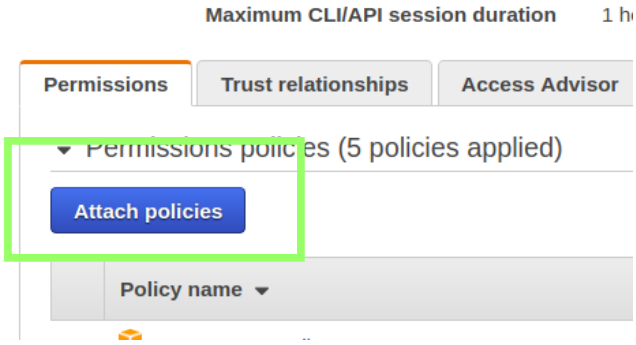
Next, using NetBeans, perform a Clean Build of the Maven project to create the function's jar file for deployment.

Now, create a new Lambda function.
In the Create a Function Wizard, for **Permissions**, Initially create the function using default permission settings.

Next, adjust the security permissions.
Under the function's **Configuration** tab, select **Permissions** from the left.

Click on the blue **Role name** link.
This will open the function's security role in the IAM role editor.

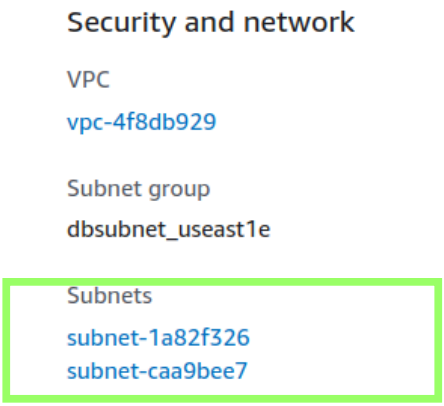
Click on the blue **Attach Policies** button and attach the following policies:
**AmazonRDSFullAccess**
**AWSLambdaVPCAccessExecutionRole**
Then close the IAM role editor and go back to the AWS Lambda GUI.



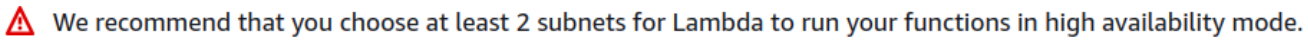

Next, configure this Lambda function to run inside the same VPC and subnet as your RDS database. If not, there will be no connectivity between Lambda and the RDS database.

Under the "**Configuration**" tab, select **VPC** on the left. Now, edit the VPC configuration. From the dropdown list, select the VPC that is labeled as "Default". Next, specify the function's subnet(s).

To check which subnets your RDS serverless database is using, navigate to RDS.
On the left hand-side select “Databases”, then select your database “tcss562”.
Scroll down to Details, and look under “Connectivity and Security”.



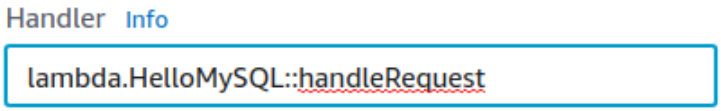

**IMPORTANT:** The Subnet for your Lambda function must be in the list of Subnets that appears here. Select **at least one subnet** for your function that is shared with the RDS database. If using the Ohio region, expect subnets to be us-east-2a, us-east-2b, or us-east-2c. Do not worry about this message if you receive it:



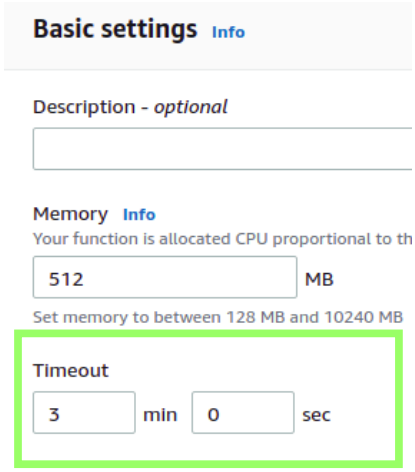

Select the default security group.

Now, Create the function.
Once the function has been created, upload the jar file.

Under “Runtime settings” set the handler:




Under the “**Configuration**” tab, under “**General configuration**” on the left:
Set the Timeout to be greater than 3 minutes. Working with RDS will require long timeouts greater than 25 seconds. This allows the serverless database time to initialize when it is revived from hibernation.

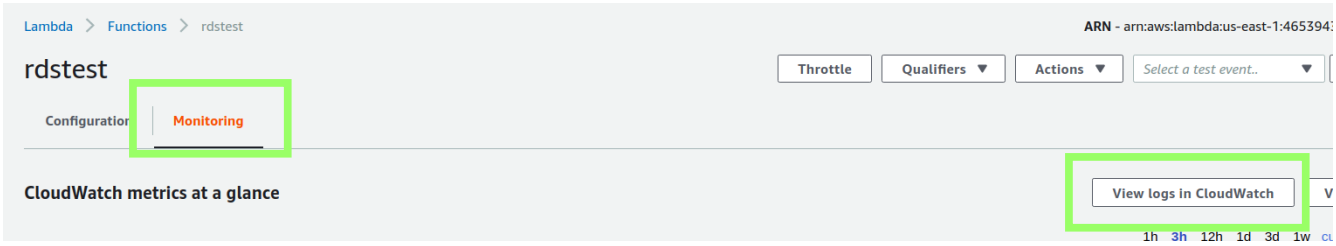



Your Lambda function should be ready to use.

Next configure callservice.sh to use the name of your newly deployed Lambda function.
Use the AWS CLI to invoke the function directly. **Using the AWS CLI to invoke Lambda directly is recommend because of the potential for long timeouts when working with RDS Aurora Serverless.**

Using callservice.sh, invoke the Lambda function to writes to the database=TEST table=mytable several times (3x-5x).
Each call to HelloMySQL will append a row to the table with the provided name.

It may be necessary to troubleshoot your Lambda function's connectivity to RDS.
From the Monitoring tab of Lambda, use the [View logs in CloudWatch] button:




7. Wait 5+ minutes. Run your callservice.sh script with the time command as follows:

```
time ./callservice.sh
```

a) How long does your Lambda function take **in seconds** after waiting for 5 minutes?
b) What is the value of the newcontainer attribute?
Does this indicate your Lambda function is warm? (YES/NO)
A "1" indicates a **COLD** container.

Run your function again with the time command.

c) How long does a second call take **in seconds**?

For Question #7, if you've created your database with anything besides 1 Aurora Compute Unit (ACU), please describe this with your answer.

8. For question #8, modify the Lambda service to return the MySQL version as a response object parameter. Add a getter/setter to Response.java for "mysqlversion". Then, add an additional SQL query to obtain the version of MySQL. Use the following SQL query:

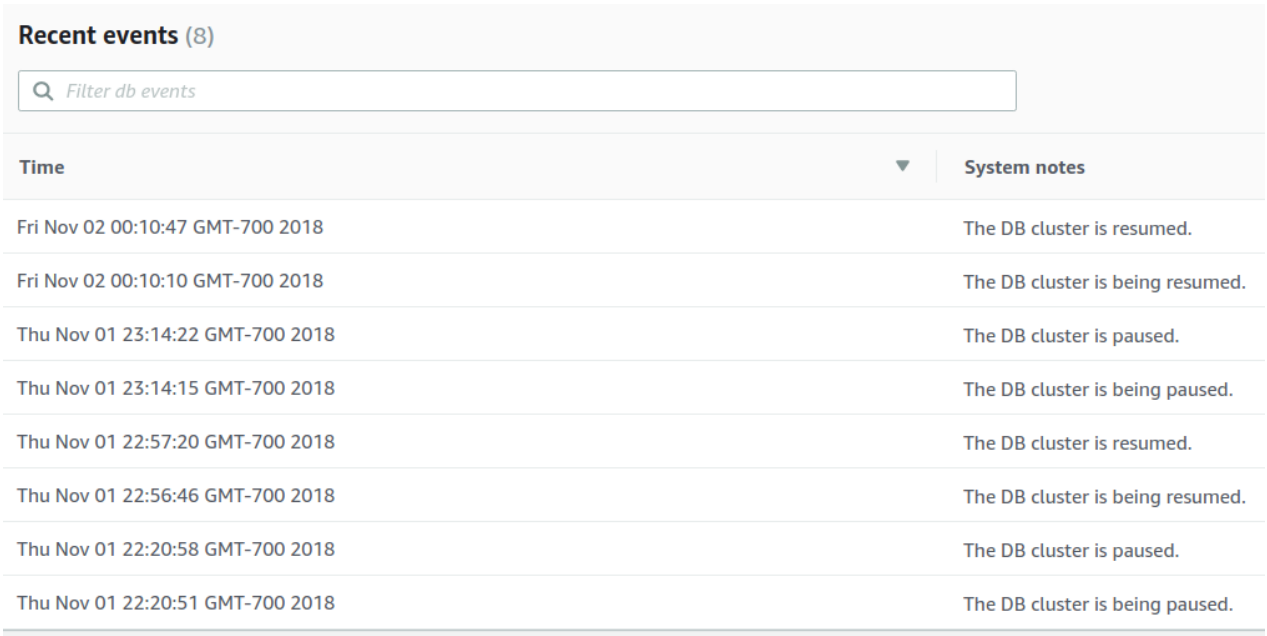
```
select version() as version;
```

With a result set, read the value from the column, and add it to the Response object.

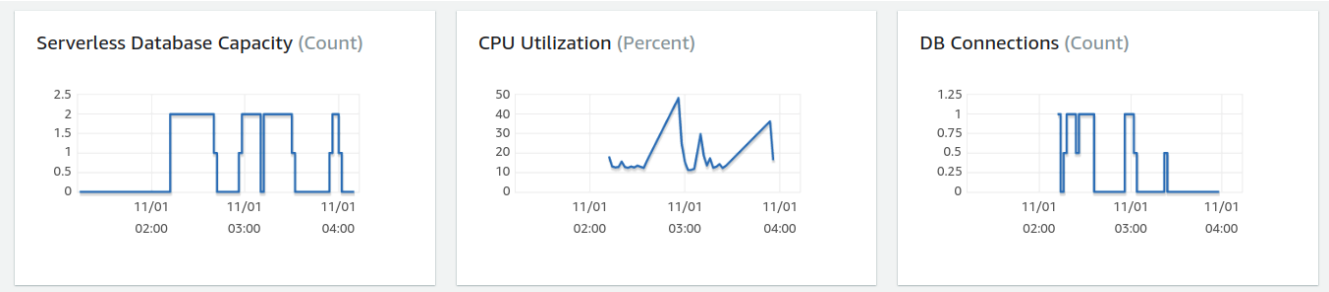
Now, using callservice.sh, run the service. Capture the complete output from the terminal as the answer for #8.

Aurora Serverless deprovisions itself after ~5 minutes of inactivity. When COLD, the first call to the serverless database extra time to "thaw" the database from hibernation.

The serverless freeze/thaw times are reported under the "Logs & events" tab on the RDS database page. From the log, previous periods of activity and hibernation can be seen.



CloudWatch graphs show RDS database resource utilization:




---

## Submitting Tutorial #6

Create a PDF file using Google Docs, MS Word, or OpenOffice. Capture answers to questions 1-8 and submit the PDF on Canvas.

Be sure to terminate EC2 instances, and destroy your RDS database once completing the tutorial.

**Related Articles providing additional background:**

Article describing use cases for when to use the SQLite database:
https://www.sqlite.org/whentouse.html

Amazon RDS Aurora MySQL 5.6 Serverless
https://aws.amazon.com/blogs/aws/aurora-serverless-ga/

Serverless DB Blog Article
https://medium.com/searce/amazon-aurora-serverless-features-limitations-glitches-d07f0374a2ab

Serverless Aurora MySQL 5.6
https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-serverless.how-it-works.html#aurora-serverless.how-it-works.auto-scaling

Research paper on AWS Aurora – Cloud Native relational database with built in read replication up to 15-nodes:
https://media.amazonwebservices.com/blog/2017/aurora-design-considerations-paper.pdf