Assignment 6: Serverless APIs with SQL, NoSQL and Cache Databases on AWS

Overview

In this assignment, you will compare three AWS data stores by building three serverless REST endpoints with **API Gateway** (HTTP API) and **AWS Lambda**:

1) PostgreSQL on **Amazon RDS**

2) Amazon **DynamoDB** (NoSQL key-value).

3) Amazon **ElastiCache** for Redis (in-memory cache).

You'll test with **Postman**, capture each database operation's elapsed time, visualize the results.

Deliverables

Each student (or group) must submit the following:

1. **Presentation Slides (10–14 slides maximum)**
   - Title slide with student name(s), course, and assignment title.
   - Clear explanation of each completed task.
   - Screenshots or videos of key steps/results.
   - Reflection question responses.
   - Lessons learned and challenges encountered.
2. **Demonstration (in-class or recorded)**
   - A 8–11 minute walkthrough of the slides and discussion of your results.
   - Optional short demo (if possible) showing the live static website or EC2 instance connection.

Presentation Guidelines

- Use professional formatting (legible fonts, consistent colors, organized layout).
- Ensure clarity: avoid overcrowding slides with text; use bullet points and visuals.
- Provide screenshots or videos as evidence of work completed.

- Keep presentations within the time limit (5–8 minutes).
- Be prepared to answer questions about the process and challenges.

Detailed Grading Rubric (100 points total)

All team members are required to contribute to the presentation. The final score will be proportionally adjusted based on the number of members who actually present. For example, if a group consists of five students but only four deliver the presentation, and the maximum score is 100 points, the adjusted score will be calculated as:

$$Adjusted\ Score = \left(\frac{4}{5}\right) \times 100 = 80$$

1. Task Completion (60 points total)

Each AWS task is evaluated based on correctness, completeness, and evidence (e.g., screenshots, output).

- Task 1 – Create Databases **(10 pts)**
- Task 2 – Common Setup (VPC Endpoints) **(7 pts)**
- Task 3 – Lambda Functions (nosql-items-fn, sql-items-fn, cache-items-fn) **(12 pts)**
- Task 4 – HTTP API & Routes (API Gateway) **(10 pts)**
- Task 5 – Test with Postman & Collect Latency **(12 pts)**
- Task 6 – Clean-Up (Manual) **(3 pts)**
- Task 7 – Reflection Questions **(6 pts – 2pts for each)**

2. Presentation Quality (30 points total)

- Clear structure (intro, results, discussion, conclusion) **(10 pts)**
- Slides visually organized and professional, for example, adding pictures, transitions, animations **(10 pts)**
- Screenshots or videos included as evidence **(5 pts)**
- Proper timing (within 8–11 min) **(5 pts)**

3. Professionalism & Delivery (10 points total)

- Confident and clear presentation delivery **(4 pts)**
- Ability to answer questions from peers/instructor **(4 pts)**
- Team collaboration and equal participation **(2 pts)**

Submission Instructions

- Submit slides (PowerPoint or PDF) to the LMS by the deadline.

- Each group will present during the next class session.

Task 1: Create Databases (RDS PostgreSQL, DynamoDB, ElastiCache Redis)

Provision the three data backends with secure networking.

Instructions:

1. RDS PostgreSQL (sandbox/free-tier template)

1. **RDS → Create database → PostgreSQL**.
   - **Templates:** Sandbox
   - **DB instance identifier:** `pg-sandbox`; **DB name:** `studentdb`.
   - **DB instance class:** `db.t3.micro`.
   - **Allocated storage:** 20 GiB (gp3/gp2).
   - **Public access: No** (private).
   - **VPC/Subnets:** use your default VPC.
   - **Security group:** create `pg-sg` that allows **inbound TCP 5432 from** `lambda-sg` (you'll make `lambda-sg`).
   - After creation, note the **RDS endpoint** (hostname).
2. **Secrets Manager** → store DB credentials if not auto-created.

2. DynamoDB

1. *DynamoDB* → *Tables* → *Create table* → name `nosql-items`.
2. Partition key `id` (String).
3. Table settings: **Default settings**.

3. ElastiCache for Redis (TLS)

1. *ElastiCache* → ***Redis OSS cache*** → *Create*.
2. **Configuration**: `Redis OSS`.
3. **Settings - Name:** `redis-db`.

Task 2: IAM Role & Common Setup

Create a reusable Lambda execution role with the minimum permissions required and a VPC Interface Endpoint for KMS.

Instructions:

Create a VPC Interface Endpoint for KMS

1. Go to **VPC → Endpoints → Create endpoint**.
2. Service category: **AWS services**.
3. For KMS:

   - Service name: `com.amazonaws.<region>.kms`.
   - Type: **Interface**.

- VPC: your RDS/Lambda VPC.
- Subnets: private subnets where your Lambda runs.
- Security groups: create new `vpce-sg` to allow HTTPS (port 443) from Lambda's SG.

4. Create a similar endpoint for: `com.amazonaws.<region>.secretsmanager`.

Why? VPC interface endpoints let your Lambda call KMS/Secrets Manager privately without needing a NAT gateway, improving security and reducing data transfer through the public Internet.

Task 3: Creating Lambda functions

Implement handlers for each data store.

1. Create nosql-items-fn Lambda

Instructions:

1. Go to **Lambda → Create function**.
2. Author from scratch:
   - Name: `nosql-items-fn`.
   - Runtime: **Python 3.x** (a supported version).
3. After creation:
   - Upload the code from `nosql_handler.zip` into the function.
   - In **Runtime settings -> Handler:** edit it to `nosql_handler.handler`.
   - In **Configuration → Environment variables**, add:
     - `TABLE_NAME` = your DynamoDB table name.
   - Attach an inline policy to your Lambda function role:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "DDBItemsRW",
      "Effect": "Allow",
      "Action": [
        "dynamodb:PutItem",
        "dynamodb:GetItem",
        "dynamodb:DescribeTable"
      ],
      "Resource": "<dynamodb-arn>"
    }
  ]
}
```

4. You can test your Lambda function with:

   ```
   {
     "version": "2.0",
     "routeKey": "POST /nosql/items",
     "rawPath": "/nosql/items",
     "requestContext": { "http": { "method": "POST" } },
     "body": "{\"id\":\"1\",\"name\":\"alice\",\"value\":42}"
   }
   ```

2. Create sql-items-fn Lambda

Instructions:

1. Go to **Lambda → Create function**.
2. Name: `sql-items-fn`.
3. Runtime: Python 3.x.
4. In **Configuration → VPC**:
   - Attach the same VPC and subnets as RDS.
   - Use the Lambda security group that is allowed to reach RDS/Endpoints.
5. Upload `sql_handler.zip` as the code.
6. In **Runtime settings -> Handler:** edit it to `sql_handler.handler.`
7. In **Environment variables**, set:
   - `SECRET_ARN` = ARN of your secret.
   - `DB_HOST` = RDS endpoint hostname (from RDS console).
   - (Optional) `DB_NAME` if you used something other than `studentdb`.
   - (Optional) `DB_PORT` if not 5432.
8. Add policy to your Lambda function IAM role to get secret and key access to PostgreSQL database:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
         "ec2:CreateNetworkInterface",
        "ec2:DescribeNetworkInterfaces",
        "ec2:DeleteNetworkInterface"
      ],
      "Resource": "*"
    },
    {
      "Sid": "ReadDbSecret",
      "Effect": "Allow",
      "Action": [
        "secretsmanager:GetSecretValue"
```

```
        ],
        "Resource": "<SECRET_ARN>"
      },
      {
        "Sid": "DecryptSecretIfCMK",
        "Effect": "Allow",
        "Action": [
          "kms:Decrypt"
        ],
        "Resource": "*",
        "Condition": {
          "StringEquals": {
            "kms:ViaService": "secretsmanager.<REGION>.amazonaws.com",
            "kms:EncryptionContext:aws:secretsmanager:arn": "<SECRET_ARN>"
          }
        }
      }
    ]
  }
```

9. Create VPC for the `sql-items-fn` Lambda function, with the same VPC and subnets as the PostgreSQL RDS database.
10. Attach `lambda-sg` security group to the Lambda function.
11. You can test your Lambda function with:

```
{
  "version": "2.0",
  "routeKey": "POST /sql/items",
  "rawPath": "/sql/items",
  "requestContext": { "http": { "method": "POST" } },
  "body": "{\"id\":\"1\",\"name\":\"alice\",\"value\":42}"
}
```

3. Create cache-items-fn Lambda

Instructions:

1. Go to **Lambda → Create function**.
2. Name: `cache-items-fn`.
3. Runtime: Python 3.x.
4. Attach to the same **VPC** and **subnets** as the Redis cluster.
5. Upload `cache_handler.zip` as the code.
6. In **Runtime settings -> Handler:** edit it to `cache_handler.handler.`
7. In **Environment variables**, set:

- `REDIS_HOST` = *primary endpoint* of your Redis cluster (without port).
- (Optional) `REDIS_PORT` = Redis port (default 6379).
- Set `REDIS_TLS`: `"true"`
- Attach an inline policy to your Lambda function role:

```
{
  "Version": "2012-10-17",
        "Statement": [
                {
                        "Effect": "Allow",
                        "Action": [
                                "ec2:CreateNetworkInterface",
                                "ec2:DescribeNetworkInterfaces",
                                "ec2:DeleteNetworkInterface"
                        ],
                        "Resource": "*"
                }
        ]
    }
```

- Create VPC for the `cache-items-fn` Lambda function, with the same VPC and subnets as the Redis-OSS cache.
- Attach `lambda-sg` security group to the Lambda function.

7. You can test your Lambda function with:

```
{
  "version": "2.0",|
  "routeKey": "POST /cache/items",
  "rawPath": "/cache/items",
  "requestContext": { "http": { "method": "POST", "path": "/cache/items" } },
  "isBase64Encoded": false,
  "body": "{\"id\":\"1\",\"name\":\"alice\",\"value\":42}"
}
```

Task 4: HTTP API (API Gateway) and Routes

Expose one unified HTTP API to all three Lambda functions.

Instruction

1. **Create an HTTP API**

   - Go to **API Gateway → HTTP APIs → Create**.
   - Choose **Build**.
   - Name: `db-api`.

2. **Create Lambda integrations**

   - For each Lambda:
     - Add an integration pointing to:
       - `sql-items-fn`
       - `nosql-items-fn`
       - `cache-items-fn`

3. **Create routes**: Create the following routes and attach them to the proper Lambda integration:

- For PostgreSQL:
    - `POST /sql/items` → `sql-items-fn`
    - `GET /sql/items` → `sql-items-fn`
- For DynamoDB:
    - `POST /nosql/items` → `nosql-items-fn`
    - `GET /nosql/items` → `nosql-items-fn`
- For Redis:
    - `POST /cache/items` → `cache-items-fn`
    - `GET /cache/items` → `cache-items-fn`

Ensure payload format is **2.0** (default for HTTP API), so that fields like `requestContext.http.method` and `rawPath` look like what the handlers expect.

Why? Using one HTTP API for all three backends gives a uniform interface and makes it easy to compare them using the same client calls.

4. **Get the base URL**

Task 5: Test with Postman & Collect Latency

Verify correctness and collect performance data.

Instruction:

1. **Create a Postman collection**

- Base URL variable: `{{baseUrl}}` = your API Invoke URL.
- Add 6 requests (as described earlier):
    - `POST /sql/items`, `GET /sql/items?id=1`
    - `POST /nosql/items`, `GET /nosql/items?id=1`
    - `POST /cache/items`, `GET /cache/items?id=1`
- Body for POST:

```
{ "id": "1", "name": "alice", "value": 42 }
```

2. **Run multiple times and record `elapsed_ms`**

- For each **database** and each **operation** (POST/GET):
    - Run the request at least **10 times**.
    - Copy the `elapsed_ms` values into a table (e.g. Excel, Google Sheets, or a CSV).

Example table structure:

Backend	Operation	Run #	elapsed_ms

postgres	create	1	12.3
postgres	create	2	10.1
...	...	...	...

Why? Repeating the measurements allows you to average out noise due to cold starts and network variability.

3. **Compute summary statistics**

For each backend and operation, calculate:

- Mean latency (average `elapsed_ms`)
- (Optional) Standard deviation

4. **Create a graph**

- Producing at least one chart, for example:
    - A bar chart with **x-axis** = backend (`postgres`, `dynamodb`, `redis`), **y-axis** = mean `elapsed_ms`, with different colors for `create` vs `get`.

You may use:

- Excel/Google Sheets
- Python (Matplotlib), R, or any tool of your choice

Task 6: Clean-Up

Before moving on to Terraform, **manually delete** all resources created to avoid ongoing charges:

- API Gateway HTTP API
- Lambda functions
- DynamoDB table
- ElastiCache Redis cluster
- RDS PostgreSQL instance
- Secrets Manager secret (optional if no longer needed)
- VPC endpoints (if created specifically for this lab)

Task 7: Answer reflection questions

Include your answers as a separate section in your presentation.

Required Question

Compare the three backends (PostgreSQL, DynamoDB, Redis) in terms of:

- Average latency for create vs get operations (based on your measurements).
- Data model flexibility (schema, indexing).
- Typical use cases where each would be the best fit.

Use your timing graph and configuration experience to justify your answer.

Selective Questions (choose one of the following four)

2. **Caching strategy:**
   Suppose your application primarily reads data that rarely changes. How would you redesign your API or architecture to leverage Redis more effectively as a cache in front of PostgreSQL or DynamoDB? What consistency or invalidation issues would you need to handle?

3. **Security and networking:**
   Reflect on storing credentials in Secrets Manager and using KMS + (optionally) VPC endpoints. What are the benefits of this approach compared to putting credentials directly in Lambda environment variables or code? Are there any drawbacks?

4. **Scalability and cost:**
   If you expect traffic to grow from tens to millions of requests per day, how do PostgreSQL, DynamoDB, and Redis compare in terms of horizontal scaling and cost model? Which combination would you recommend and why?

5. **Serverless vs. managed but not serverless:**
   Lambda + API Gateway are serverless (you don't manage servers), whereas RDS and ElastiCache are managed but still capacity-provisioned. Discuss pros and cons of this mix. Would you consider Aurora Serverless or DynamoDB Streams/Lambda integration to make the architecture "more serverless"? Explain.

Good luck!