Lab: High Availability and Disaster Recovery with AWS Serverless Architecture

Lab Overview

This lab demonstrates how to build a highly available and disaster-resilient application using AWS serverless services. The solution includes: - **DynamoDB Global Tables** for cross-region replication. - **AWS Lambda** (Python) for serverless compute. - **API Gateway** to expose REST APIs. - **Amazon Route 53** for DNS-based failover. - A **frontend website** (HTML + CSS + JavaScript) to interact with the APIs.

Step 1: Set Up DynamoDB Global Tables

DynamoDB Global Tables provide automatic multi-region replication.

Instructions:

1. Go to the **AWS Management Console**.
2. Navigate to **DynamoDB**.
3. Create a new table in the primary region (e.g., `us-east-1`):

- **Table name**: `HighAvailabilityTable`.
- **Partition key**: `ItemId` (String).

4. After creating the table, enable **Global Tables**:

- Go to the **Global Tables** tab.
- Click **Create replica** and select the secondary region (e.g., `us-west-2`).

5. Wait for the replication to complete.

Step 2: Create IAM Roles for Lambda Functions

Lambda functions need permissions to interact with DynamoDB.

Instructions:

1. Go to the **IAM Console**.
2. Create a new role:

- **Role name**: `LambdaDynamoDBRole`.
- Attach the following policies:
  - `AmazonDynamoDBFullAccess` (for simplicity; restrict permissions in production).
  - `AWSLambdaBasicExecutionRole` (for CloudWatch logging).

3. Note the role ARN; you'll need it when creating the Lambda functions.

Step 3: Create Lambda Functions in Both Regions

We'll create two Lambda functions in Python: one for reading data and one for writing data.

Code for Lambda Functions:

1. **Read Function** (`read_function.py`):

```python
import json
import boto3
from boto3.dynamodb.conditions import Key

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('HighAvailabilityTable')

def lambda_handler(event, context):
    try:
        response = table.scan()
        items = response['Items']
        return {
            'statusCode': 200,
            'headers': {
                'Content-Type': 'application/json',
                'Access-Control-Allow-Origin': '*',  # Enable CORS
            },
            'body': json.dumps(items)  # Return the items array
        }
    except Exception as e:
        return {
            'statusCode': 500,
            'headers': {
                'Content-Type': 'application/json',
                'Access-Control-Allow-Origin': '*',  # Enable CORS
            },
            'body': json.dumps({'error': str(e)})
        }
```

2. **Write Function** (`write_function.py`):

```python
import json
import boto3

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('HighAvailabilityTable')

def lambda_handler(event, context):
    try:
        body = json.loads(event['body'])
        item_id = body['ItemId']
        data = body['Data']

        table.put_item(Item={'ItemId': item_id, 'Data': data})
        return {
            'statusCode': 200,
            'body': json.dumps({'message': 'Item saved successfully'})
        }
    except Exception as e:
        return {
            'statusCode': 500,
            'body': json.dumps({'error': str(e)})
        }
```

Instructions:

1. Go to the **AWS Lambda** console.
2. Create a new function in the primary region:

- **Name**: `ReadFunction`.
- **Runtime**: Python 3.9.
- **Role**: Choose the `LambdaDynamoDBRole` created earlier.
- Paste the `read_function.py` code.

3. Repeat for the `WriteFunction` using the `write_function.py` code.
4. Deploy the same functions in the secondary region.

Step 4: Set Up API Gateway in Both Regions

Create REST APIs to expose the Lambda functions.

Instructions:

1. Go to the **API Gateway** console.
2. Create a new REST API:

- **Name**: `HighAvailabilityAPI`.
- Create two resources: `/read` and `/write`.
- Enable CORS (Cross Origin Resource Sharing).
- Link `/read` to `ReadFunction` with a `GET` method.
- Link `/write` to `WriteFunction` with a `POST` method.
- Use proxy integrations.

3. Deploy the API to a new stage (e.g., `prod`).
4. Note the API Gateway endpoint URL.
5. Repeat the same setup in the secondary region.

Step 5: Create ACM certificates and custom domain for the APIs

We need a custom domain for our APIs along with SSL/TLS certificates before we can create the Route 53 records.

Instructions:

1. Go to the **Certificate Manager** console.
2. Request a certificate.
3. Use the domain name "api.".
4. Use DNS validation.
5. Go to the **API Gateway** console and click "Custom domain names".
6. Click "Add domain name" and enter the same subdomain as above.
7. Select the appropriate certificate and create the domain name.
8. Create an API mapping selecting your API and stage.
9. Repeat the same steps for the second Region.

Step 6: Set Up Route 53 DNS Name

Now that the APIs exist, we'll create a DNS name (e.g., `api.example.com`) that points to the active region.

Create the health checks:

1. Go to the **Route 53** console.
2. Go to "Health checks".
3. Create health checks as follows:

- Name: Primary / Secondary
- Resource: Endpoint
- Specify endpoint by: Domain name
- Domain name: Use HTTPS and enter the API invoke URL (without the HTTPS://) and with /read on the end. E.g. `w0oebdxxzh.execute-api.us-east-1.amazonaws.com/pro`

Create the records:

1. Go to the **Route 53** console.
2. Create a new record set:

- **Name**: `api.example.com`.
- **Type**: `A` / Alias (IPv4 address).
- **Routing policy**: `Failover`.
- **Primary endpoint**: API Gateway URL for the primary region.
- **Secondary endpoint**: API Gateway URL for the secondary region.
- **Health check**: Select the appropriate health checks.

3. Note the DNS name (e.g., `api.example.com`).

Step 7: Create the Frontend Website

The frontend will use HTML, Bootstrap for styling, and JavaScript to interact with the API.

Code for Frontend (`index.html`):

```html
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>High Availability App</title>
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="st
</head>
<body class="bg-light">
<div class="container mt-5">
    <h1 class="text-center mb-4">High Availability App</h1>
    <div class="card mb-4">
        <div class="card-body">
            <h2 class="card-title">Add Data</h2>
            <div class="mb-3">
                <input type="text" id="itemId" class="form-control" placeholder="Item ID">
            </div>
            <div class="mb-3">
                <input type="text" id="itemData" class="form-control" placeholder="Item Data
            </div>
            <button class="btn btn-primary" onclick="writeData()">Save</button>
        </div>
    </div>
    <div class="card">
        <div class="card-body">
            <h2 class="card-title">Data List</h2>
            <ul id="dataList" class="list-group"></ul>
        </div>
    </div>
</div>

<script>
    const apiUrl = 'https://api.<YOUR-DOMAIN>'; // Use Route 53 DNS name

    async function writeData() {
        const itemId = document.getElementById('itemId').value;
        const itemData = document.getElementById('itemData').value;

        const response = await fetch(`${apiUrl}/write`, {
            method: 'POST',
            body: JSON.stringify({ ItemId: itemId, Data: itemData }),
            headers: { 'Content-Type': 'application/json' },
        });

        const result = await response.json();
        alert(result.message || result.error);
        readData();
    }

    async function readData() {
        const response = await fetch(`${apiUrl}/read`);
        const data = await response.json();
        const dataList = document.getElementById('dataList');
        dataList.innerHTML = '';

        data.forEach(item => {
            const li = document.createElement('li');
            li.className = 'list-group-item';
            li.textContent = `ID: ${item.ItemId}, Data: ${item.Data}`;
            dataList.appendChild(li);
        });
    }

    readData(); // Load data on page load
</script>
</body>
</html>
```

Instructions:

1. Replace `api.example.com` with your Route 53 DNS name.
2. Host the `index.html` file on S3 or serve it via API Gateway.

Step 8: Test the Failover Mechanism

1. **Simulate a Failure**:

- Delete the API Gateway in the primary region.
- Route 53 health checks will detect the failure and route traffic to the secondary region.

2. **Verify the Frontend**:

- The frontend will continue to work seamlessly, as it uses the Route 53 DNS name to connect to the active region.

Final Notes

- **DNS Propagation**: Route 53 DNS changes may take a few minutes to propagate.
- **Health Check Frequency**: Configure health checks to match your failover requirements (e.g., 30-second intervals).