Event-Driven cloud Architecture, CQRS and Event Sourcing for User Management

2019/08/09

Description

Documentation about cloud architectures that use an Event-Driven approach and implement CQRS and Event Sourcing for User Management.

Table of contents

M//17

List of figures

1   Introduction

1.1   Project purpose

This project explains how to build Event-Driven architectures, CQRS and Event Sourcing for user management, showing how should be implemented, extended, strengths and weaknesses. The application for user management implements an authentication function and the CRUD operations for each aggregates, which are: users, roles, authorizations and groups.

1.2   References

- **What do you mean by "Event-Driven"? - Martin Fowler**: martinfowler.com/articles/201701-event-driven.html;

- **CQRS - Martin Fowler**: martinfowler.com/bliki/CQRS.html;

- **Event Sourcing - Martin Fowler**: martinfowler.com/eaaDev/EventSourcing.html;

- **AWS Lambda documentation**: https://docs.aws.amazon.com/lambda/index.html;

- **AWS DynamoDB documentation**: https://docs.aws.amazon.com/dynamodb/index.html;

- **AWS API gateway documentation**: https://docs.aws.amazon.com/apigateway/index.html;

- **AWS SQS documentation**: https://docs.aws.amazon.com/sqs/index.html;

- **Serverless Framework**: https://serverless.com/;

2 Amazon Web Services

Amazon Web Services (AWS) is a subsidiary of Amazon that provides on-demand cloud computing platforms to individuals, companies, and governments, on a pay-as-you-go basis. These cloud computing web services provide a set of primitive abstract technical infrastructure and distributed computing building blocks and tools. AWS's version of virtual computers emulate most of the attributes of a real computer including, hardware central processing units and graphics processing units, local/RAM memory, hard-disk/SSD storage; a choice of operating systems; networking; and pre-loaded application software such as web servers and databases.

2.1 Lambda

AWS Lambda is an event-driven, serverless computing platform provided by Amazon. It is a computing service that runs code in response to events and automatically manages the computing resources required by that code. The purpose of Lambda is to simplify building smaller, on-demand applications that are responsive to events and new information. AWS targets starting a Lambda instance within milliseconds of an event. Node.js, Python, Java, Go, Ruby and C# through .NET Core are all officially supported.

In this project, lambda functions are the computing core for the execution of all commands and are written in Node.js.

2.1.1 Write mode Lambda functions

2.1.1.1 PushOperationAggregateToSQS

Those functions starts the flow and triggered when fetching the corresponding API gateway URL with a POST request like this:


```
fetch("https://domain/resourceAPIpath", {
  method: "post",
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json'
  },

  body: JSON.stringify({
    "attribute1": "value1",
    "attribute2": "value2"
  })
});
```

Fig. 1: Fetch POST API

Once triggered, these functions retrieve the event and put it in the *MessageBody* parameter. The event object is not changed, instead in the user functions which encrypt the password before sending the message to corresponding SQS queue.
Example:

```
1  module.exports.pushUpdateRoleToSQS = (event, context, callback) => {
2      const AWS = require('aws-sdk');
3      const SQS = new AWS.SQS();
4      const stringedEvent = JSON.stringify(event);
5
6      const params = {
7          MessageBody: stringedEvent,
8          QueueUrl: "https://yourDomain/updateRoleQueue"
9      };
10
11     SQS.sendMessage(params, function(err, data){
12         if(err){
13             console.log(err);
14             callback(null, err);
15         }
16         else
17             callback(null, "Role event pushed to SQS");
18     });
19 };
```

2.1.1.2 CommandOperationAggregate

These functions validate the values of the attributes before storing the event in the *eventStore* and triggered when a new event arrives to the corresponding queue.
If you have to check for duplicated attributes in the database, the function must be marked *async* because it has to wait for the result of the check operation.
Example:

```
1  module.exports.commandCreateRole = async (event, context, callback) => {
2      const utils = require('./utils.js');
3
4      const stringedEvent = event.Records[0].body.toString('utf-8');
5      const eventParsed = JSON.parse(stringedEvent);
6      const stringedBody = JSON.stringify(eventParsed);
7      const bodyParsed = JSON.parse(stringedBody);
8      const check = bodyParsed.body;
9
10     const checkNameParams = {
11         TableName: 'role',
12         ExpressionAttributeNames:{
13             "#rolename": "name"
14         },
15         ProjectionExpression: "#rolename",
16         FilterExpression: "#rolename = :checkname",
17         ExpressionAttributeValues: {
```

```
18          ":checkname": check.name
19       }
20
21    const nameAlreadyExists = await utils.asyncCheckScanDB(checkNameParams);
22
23    if ((check.roleId == "" || check.name == "" || check.desc == "") || ↩
          nameAlreadyExists)
24        callback(null, "Name already exists or empty attributes");
25    else{
26        utils.storeEvent("role", "executeCreateRoleQueue", bodyParsed.body);
27        callback(null, "Role event stored");
28    }
29 };
```

2.1.1.3   Mediator

This function catch the *DynamoDB* event and triggered when a change occurs in a certain table.
You have to be careful that the *DynamoDB* events are mapped using the char type attribute value. This is an example:

```
{
    "eventId": {
        "S": "bf6dffb9-72d8-ae5f-21fa-56dd6a26d572"
    },
    "payload": {
        "M": {
            "auth": {
                "S": "{\n\t\"Authorizations\":  [\n\t\t\"FullAccess\" ] }"
            },
            "roleId": {
                "S": "63c471c1-e5c7-09d0-ea8a-f20b27a0575c"
            },
            "name": {
                "S": "Admin"
            },
            "desc": {
                "S": "Full access to all resources"
            }
        }
    },
    "aggregate": {
        "S": "role"
    },
    "executionQueue": {
        "S": "executeCreateRoleQueue"
    },
    "timestamp": {
        "N": "1563358792450"
    }
}
```

Fig. 2: DynamoDB event object

You can use the *"AWS.DynamoDB.Converter"* module to parse a DynamoDB event object.

Fig. 3: Parsed DynamoDB event object

After that, the mediator retrieves the *executionQueue* parameter from the event object, the payload event passed with the *MessageBody* and then sends the execution message to corresponding SQS queue.

2.1.1.4 OperationAggregate

These functions execute a single operation using the event payload and triggered when new event arrives to the corresponding execution queue.
Example:

```javascript
module.exports.createRole = async (event, context, callback) => {
    const AWS = require('aws-sdk');
    const dynamoDb = new AWS.DynamoDB.DocumentClient();

    const stringedBody = event.Records[0].body.toString('utf-8');
    const parsedBody = JSON.parse(stringedBody);

    const params = {
        TableName: 'role',
        Item: parsedBody
    };

    await dynamoDb.put(params, (err, data) => {
        if (err){
            console.log(err);
            callback(null, err);
        }
    else
        callback(null, "Role created");
    }).promise();
};
```

2.1.1.5   Recovery

This function allows you to restore the system state starting from a given timestamp by replaying all the events into the *eventStore* table which were stored after that time.

Is important to re-execute the events one by one and in the correct order(from the oldest one); for this purpose the recovery function implements a sorting algorithm that retrieves an array of events and sorts them by timestamp, and an *async* function which sends every event to the corresponding execution queue.

2.1.2   Read mode Lambda functions

2.1.2.1   ReadOperationAggregate

These functions works in the read side of the architecture; they query the database to retrieve the informations needed without passing throw a SQS queue or a mediator. This kind of events aren't stored into the *eventStore* because they don't change the current system state; this Lambda functions fetch the URL of a GET endpoint using a GET request and listen for a response result.


```
fetch("https://domain/resourceAPIpath" [+ queryStringParam] {
  method: "GET",
  headers: {
    'Content-Type': 'application/json'
  }
});
```

Fig. 4: Fetch GET API

Example:

```
1  module.exports.getAllRoles = (event, context, callback) => {
2     const AWS = require('aws-sdk');
3     const dynamoDb = new AWS.DynamoDB.DocumentClient();
4
5     const params = {
6        TableName: 'role',
7        ExpressionAttributeNames:{
8        "#rolename": "name"
9        },
10       ProjectionExpression: "#rolename"
11    };
12
13    dynamoDb.scan(params, (err, data) => {
14       const stringedData = JSON.stringify(data);
15       if (err){
16          console.log(err);
17          callback(null, err);
18       }
```

```
19        else{
20            if(data.Count == 0){
21                console.log("Role not found");
22                callback(null, "Role not found");
23            }
24            else{
25                const response = {
26                    statusCode: 200,
27                    headers: {
28                        'Content-Type': 'application/json',
29                        'Access-Control-Allow-Origin': '*',
30                        'Access-Control-Allow-Credentials': true
31                    },
32                    body: stringedData
33                };
34                callback(null, response);
35            }
36        }
37    });
38 };
```

2.1.3 CloudWatch Logs

Amazon CloudWatch is a monitoring and management service built for developers, system operators, site reliability engineers and IT managers. CloudWatch provides you with data and actionable insights to monitor your applications, understand and respond to system-wide performance changes, optimize resource utilization, and get a unified view of operational health. CloudWatch collects monitoring and operational data in the form of logs, metrics, and events.
This tool can be very helpful to debug your Lambda functions and to understand what happens to your system.

2.2 DynamoDB

Amazon DynamoDB is a fully managed proprietary NoSQL database service that supports key-value and document data structures and is offered by Amazon Web Services. In this project the DynamoDB instance is used for two purpose: it is used to store events and to keep updated the aggregates views.
The tables are the following:

- **eventStore**: to keep track of the occurred events;

- **user**: to store users info;

- **role**: to store roles info;

- **authorization**: to store authorizations info;

- **group**: to store groups info.

2.3 API Gateway

Amazon API Gateway is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure API at any scale. You can create REST and WebSocket API that act as a "front door" for applications to access data, business logic, or functionality from your backend services.

In this project, API endpoints starting the execution flow of a write mode function through a POST request, pushing a new event into the corresponding queue; in fact you have to provide an API endpoint for each function. On the other side, to query the database you have to reach the endpoint through a GET request to get a result response.

2.4 Simple Queue Service

Simple Queue Service(SQS) is a distributed message queuing service introduced by Amazon. It supports programmatic sending of messages via web service applications as a way to communicate over the Internet. SQS is intended to provide a highly scalable hosted message queue that resolves issues arising from the common producer-consumer problem or connectivity between producer and consumer.

In this project, for each function you have to use two queues:

- **OperationAggregateQueue**: this queues receive new events from an API endpoint and trigger the corresponding *commandOperationAggregate* function;

- **ExecuteOperationAggregateQueue**: this queues receive new events from the *mediator* function and trigger the corresponding *operationAggregate* function.

3 Serverless Framework

3.1 Description

The Serverless Framework is a free and open-source web framework written using Node.js. Serverless is the first framework that was originally developed for building applications exclusively on AWS Lambda, a serverless computing platform provided by Amazon as a part of the Amazon Web Services. Currently, applications developed with Serverless can be deployed to other function as a service providers, including Microsoft Azure with Azure Functions, IBM Bluemix with IBM Cloud Functions based on Apache OpenWhisk, Google Cloud using Google Cloud Functions, Oracle Cloud using Oracle Fn, Kubeless based on Kubernetes, Spotinst and Webtask by Auth0.

A Serverless app can simply be a couple of lambda functions to accomplish some tasks, or an entire back-end composed of hundreds of lambda functions. Serverless currently supports Node.js and Python runtime.

3.2 Serverless.yml

One advantage of using this tool is the capability to deploy every time an entire serverless system based on cloud providers, in this case AWS. You can define a *serverless.yml* configuration file which contains all the informations about your service. You don't need to manually create the resources you need in a project, like database tables, queues, API endpoints and Lambda functions. You just have to write the code and then deploying your app.

Example of serverless.yml configuration for User Management:

- Name of your services and list of plugins: the *split-stack* plugin is useful when you reach the limit of 200 resources to deploy because it automatically split your resources using nested stacks;

Fig. 5: Plugins - serverless.yml

- About your cloud provider: you can also define your IAM role authorization policies;

Fig. 6: Provider info - serverless.yml

- DynamoDB tables parameters: you have to define the name of the table, name and type of the key and the previsioned throughput;

Fig. 7: DynamoDB tables - serverless.yml

- Lambda function and its handler: you can also define which events trigger that lambda. In this case the trigger event is when new object arrives into the specified queue;

Fig. 8: Lambda triggered by SQS event - serverless.yml
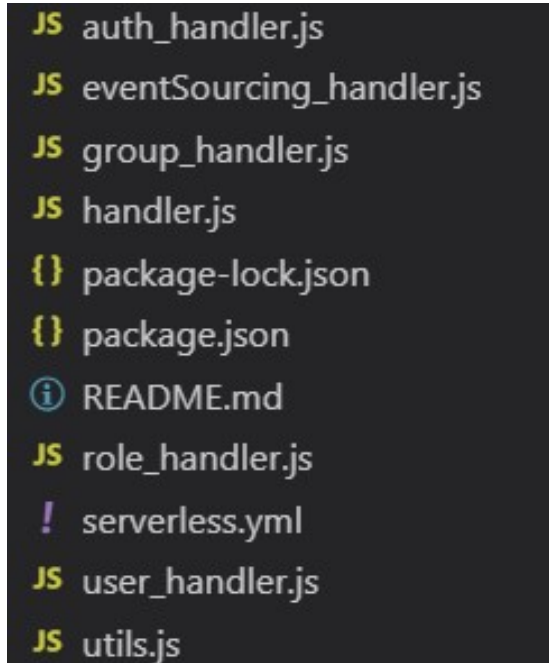
- You can define an API gateway endpoint which trigger the corresponding function: you can also use a custom request template.

Fig. 9: Lambda triggered by GET endpoint - serverless.yml

3.3   Project's root

The main project's handler is composed of the other handlers, one per aggregate and another one to handle event sourcing functions. In this way the responsibilities are restricted to every type of aggregate.

Fig. 10: Project's root

4　CQRS

4.1　Architecture overview

4.2　Write model

4.3　Read model

5   User Management

5.1   Aggregates

5.1.1   User

- **userId**: this attribute must be unique and refers to the *user_ id* key in the corresponding *Auth0* users table;

- **firstName**: user's first name;

- **lastName**: user's last name;

- **date**: user's birth date;

- **email**: this attribute must be unique and respect the right format;

- External links:

  - **role**: user's role;

  - **group**: user's belonging group.

5.1.2   Role

- **roleId**: this attribute is generated with a UUID function;

- **name**: this attribute must be unique;

- **desc**: role's description;

- External links:

  - **auth**: JSON array of role's authorizations.

5.1.3   Authorization

- **authId**: this attribute is generated with a UUID function;

- **name**: this attribute must be unique;

- **desc**: auth's description.

5.1.4   Group

- **groupId**: this attribute is generated with a UUID function;

- **name**: this attribute must be unique;

- **desc**: group's description.

5.2   Admin side

5.2.1   Authentication

The authentication feature si provided by a third part provider, Auth0. In the admin side only admin users can log in; the user must be in the corresponding application's database on Auth0.

5.2.2   Use cases

:

- Creation of new aggregate object;

- Reading of aggregate object's info;

- Update of aggregate object;

- Delete an aggregate object;

- Recover the system state from a specific timestamp.

5.3   User side

5.3.1   Authentication

The authentication feature si provided by a third part provider, Auth0. In the user side only users can log in; it can sign in with email and password or using Google's authentication or just log in using its credentials.

5.3.2   Use cases

:

- Filling the users form to sign in into the User Management app;

- Update user's info;

- Reading user's info.

6 Event Sourcing

6.1 Event store

7 Extension points

7.1 New aggregates

7.2 New write functions

7.3 New read functions

8 Setup