



SINCE TOMORROW

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

2019/08/09

About

Company	Molo17 S.r.l.
Author	Lorenzo Busin
State	Approved
Use	Internal
Email	lorenzo.busin@gmail.com

Description

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

Table of contents

1	Introduction	4
1.1	Project purpose	4
1.2	References	4
2	Amazon Web Services	5
2.1	Lambda	5
2.1.1	Write mode Lambda functions	5
2.1.1.1	PushOperationAggregateToSQS	5
2.1.1.2	CommandOperationAggregate	7
2.1.1.3	Mediator	8
2.1.1.4	OperationAggregate	10
2.1.1.5	Recovery	11
2.1.2	Read mode Lambda functions	12
2.1.2.1	ReadOperationAggregate	12
2.1.3	CloudWatch Logs	13
2.2	DynamoDB	14
2.3	API Gateway	14
2.4	Authorizer	14
2.5	Simple Queue Service	15
3	Serverless Framework	16
3.1	Description	16
3.2	Serverless.yml	16
3.3	Project's root	20
4	CQRS	21
4.1	Architecture overview	22
4.2	Write model	23
4.3	Read model	23
5	User Management	24
5.1	Aggregates	24
5.1.1	User	24
5.1.2	Role	24
5.1.3	Authorization	24
5.1.4	Group	24
5.2	Admin side	25
5.2.1	Authentication	25
5.2.2	Use cases	25
5.3	User side	26
5.3.1	Authentication	26
5.3.2	Use cases	27
6	Event Sourcing	29
6.1	Event store	29

6.1.1	Event object	29
7	Extension points	31
7.1	New aggregates	31
7.2	New write functions	31
7.3	New read functions	31
8	Setup	32
9	Upgrades	33

List of figures

1	Fetch POST API	5
2	Lambda: pushOperationAggregateToSQS	6
3	Lambda: commandOperationAggregate	7
4	DynamoDB event object	8
5	Parsed DynamoDB event object	9
6	Lambda: mediator	9
7	Lambda: operationAggregate	10
8	Lambda: recovery	11
9	Fetch GET API	12
10	Lambda: readOperationAggregate	13
11	Authorizer workflow	15
12	Plugins - serverless.yml	17
13	Provider info - serverless.yml	17
14	DynamoDB tables - serverless.yml	18
15	Lambda triggered by SQS event - serverless.yml	18
16	Lambda triggered by GET endpoint - serverless.yml	19
17	Project's root	20
18	Architecture overview	22
19	Write model sequence diagram	23
20	Read model sequence diagram	23
21	Admin use cases	25
22	User use cases	27
23	DynamoDB eventStore example item	30

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

- **CQRS - Martin Fowler:**
martinfowler.com/bliki/CQRS.html;
- **Event Sourcing - Martin Fowler:**
martinfowler.com/eaaDev/EventSourcing.html;
- **AWS Lambda docs:**
<https://docs.aws.amazon.com/lambda/index.html>;
- **AWS DynamoDB docs:**
<https://docs.aws.amazon.com/dynamodb/index.html>;
- **AWS API Gateway docs:**
<https://docs.aws.amazon.com/apigateway/index.html>;
- **AWS SQS docs:**
<https://docs.aws.amazon.com/sqs/index.html>;
- **Serverless Framework:**
<https://serverless.com/>;
- **Auth0 docs:**
<https://auth0.com/docs>.

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 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 are triggered when fetching the corresponding API gateway URL with a POST request like this:

```
fetch(linkCreateUserAPI_POST, {
  method: 'post',
  headers: {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer ' + localStorage.getItem('id_token')
  },
  body: JSON.stringify({
    "attribute1": "value1",
    "attribute2": "value2"
  })
}).catch(function(error){
  showError(error);
});
```

Fig. 1: Fetch POST API

Once triggered, these functions retrieve the event and put it in the *MessageBody* parameter. The event object doesn't change, it just stringified before sending the message to corresponding SQS queue.

Example:

```
module.exports.pushCreateUserToSQS = async (event, context, callback) => {
  const utils = require('./utils.js');
  const params = {
    MessageBody: JSON.stringify(event),
    QueueUrl:
      "https://sqs.eu-central-1.amazonaws.com/582373673306/createUserQueue"
  };
  const res = await utils.pushToSQS(params);
  callback(null, res);
};
```

Fig. 2: Lambda: pushOperationAggregateToSQS

2.1.1.2 CommandOperationAggregate

These functions validate the values of the attributes before storing the event in the *eventStore* and are 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.

In this example, the function checks for a duplicated *userId* or email address and if the attributes value is empty, else the event is stored.

```
module.exports.commandCreateUser = async (event, context, callback) => {
  const utils = require('./utils.js');
  const stringedEvent = event.Records[0].body.toString('utf-8');
  const eventParsed = JSON.parse(stringedEvent);
  const stringedBody = JSON.stringify(eventParsed.body);
  const eventToCheck = JSON.parse(stringedBody);
  const checkIdParams = { //params to check for duplicated userId
    TableName: 'user',
    ProjectionExpression: "userId",
    FilterExpression: "userId = :checkId",
    ExpressionAttributeValues: {
      ":checkId": eventToCheck.userId
    }
  };
  const userIdAlreadyExists = await utils.asyncCheckScanDB(checkIdParams);
  if(userIdAlreadyExists)
    callback(null, "userId already exists");
  const checkEmailParams = { //params to check for duplicated email
    TableName: 'user',
    ProjectionExpression: "email",
    FilterExpression: "email = :checkEmail",
    ExpressionAttributeValues: {
      ":checkEmail": eventToCheck.email
    }
  };
  const emailAlreadyExists = await utils.asyncCheckScanDB(checkEmailParams);
  if(emailAlreadyExists)
    callback(null, "Email already exists");
  if(eventToCheck.userId == "" || eventToCheck.firstName == "" ||
    eventToCheck.lastName == "" || eventToCheck.date == "" ||
    eventToCheck.role == "" || eventToCheck.group == ""){
    callback(null, "Empty attributes");
  }
  else{
    utils.storeEvent("user", "executeCreateUserQueue", eventToCheck);
    callback(null, "User event stored");
  }
};
```

Fig. 3: Lambda: commandOperationAggregate

2.1.1.3 Mediator

This function catch the *DynamoDB* event and is 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. 4: DynamoDB event object

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

```

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

```

Fig. 5: Parsed DynamoDB event object

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

```

module.exports.mediator = (event, context, callback) => {
  const AWS = require('aws-sdk');
  const SQS = new AWS.SQS();
  const parser = AWS.DynamoDB.Converter; //module to parse dynamodb objects
  try{
    var parsedEvent = parser.unmarshall(event.Records[0].dynamodb.NewImage);
  }catch (err) {
    console.log(err);
    callback(null, err);
  }
  const params = { //get the SQS params
    MessageBody: JSON.stringify(parsedEvent.payload),
    QueueUrl: "https://sqs.eu-central-1.amazonaws.com/582373673306/" +
      parsedEvent.executionQueue
  };
  SQS.sendMessage(params, function(err,data){ //push to SQS
    if(err){
      console.log(err);
      callback(null, err);
    }
    else
      callback(null, "Execution event pushed to SQS");
  });
};

```

Fig. 6: Lambda: mediator

2.1.1.4 OperationAggregate

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

Example:

```
module.exports.createUser = 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: 'user',
    Item: parsedBody
  };
  await dynamoDb.put(params, (err, data) => {
    if (err){
      console.log(err);
      callback(null, err);
    }
    else
      callback(null, "User created");
  }).promise();
};
```

Fig. 7: Lambda: operationAggregate

2.1.1.5 Recovery

This function allows you to rebuild 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.

```
module.exports.recovery = (event, context, callback) => {
  const AWS = require('aws-sdk');
  const dynamoDb = new AWS.DynamoDB.DocumentClient();
  const utils = require('./utils.js');
  const queryParams = {
    TableName: 'eventStore',
    ExpressionAttributeNames: {
      "#eventtimestamp": "timestamp", //timestamp is a reserved keyword
      "#eventaggregate": "aggregate" //aggregate is a reserved keyword
    },
    ProjectionExpression: "#eventtimestamp, #eventaggregate, payload,
      executionQueue",
    FilterExpression: "#eventtimestamp >= :timest",
    ExpressionAttributeValues: {
      ":timest": parseInt(event.body.timestamp, 10)
    }
  };
  dynamoDb.scan(queryParams, (err, data) => {
    if (err)
      callback(null, err);
    else {
      if(data.Count == 0)
        callback(null, "Events not found");
      else {
        const stringedData = JSON.stringify(data);
        const parsedData = JSON.parse(stringedData);
        var events = parsedData.Items;
        events.sort((a, b) => { //order events by timestamp from the oldest
          var a1 = a.timestamp, b1 = b.timestamp;
          if (a1 < b1) return -1;
          if (a1 > b1) return 1;
          return 0;
        });
        utils.asyncPushToExecutionQueue(events);
        callback(null, "Recovering...");
      }
    }
  });
};
```

Fig. 8: Lambda: recovery

2.1.2 Read mode Lambda functions

2.1.2.1 ReadOperationAggregate

These functions work in the read side of the architecture; they query the database to retrieve the information needed without passing through 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 function fetches the URL of a GET endpoint using a GET request and listens for a response result.

```
fetch(linkUserAPI_GET, {
  method: "get",
  headers: {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer ' + localStorage.getItem('id_token')
  }
}).then(function(response){
  return response;
}).catch(function(error){
  showError(error);
});
```

Fig. 9: Fetch GET API

Example:

```
module.exports.readUser = (event, context, callback) => {
  const AWS = require('aws-sdk');
  const dynamoDb = new AWS.DynamoDB.DocumentClient();
  const stringedEvent = JSON.stringify(event);
  const parsedEvent = JSON.parse(stringedEvent);
  const params = { //get user by userId
    TableName: 'user',
    Key: {
      "userId": parsedEvent.userId
    },
    KeyConditionExpression: "userId = :id",
    ExpressionAttributeValues: {
      ":id": parsedEvent.userId
    }
  };
  dynamoDb.get(params, (err, data) => {
    const stringedData = JSON.stringify(data);
    if (err)
      callback(null, err);
    else{
      if(data.Count == 0)
        callback(null, "User not found");
      else {
        const response = {
          statusCode: 200,
          headers: {
            'Content-Type': 'application/json',
            'Access-Control-Allow-Origin': '*',
            'Access-Control-Allow-Credentials': true
          },
          body: stringedData
        };
        callback(null, response);
      }
    }
  });
};
```

Fig. 10: Lambda: readOperationAggregate

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 Authorizer

A Lambda authorizer (also known as a custom authorizer) is an API Gateway feature that uses a Lambda function to control access to your API.

A Lambda authorizer is useful if you want to implement a custom authorization scheme that uses a bearer token authentication strategy or that uses request parameters to determine the caller’s identity.

When a client makes a request to one of your API’s methods, API Gateway calls your Lambda authorizer, which takes the caller’s identity as input and returns an IAM policy as output.

There are two types of Lambda authorizers:

- A token-based Lambda authorizer (also called TOKEN authorizer) receives the caller’s identity in a bearer token, such as a JSON Web Token (JWT);

- A request parameter-based Lambda authorizer (also called a REQUEST authorizer) receives the caller's identity in a combination of headers, query string parameters, stage variables, and context variables.

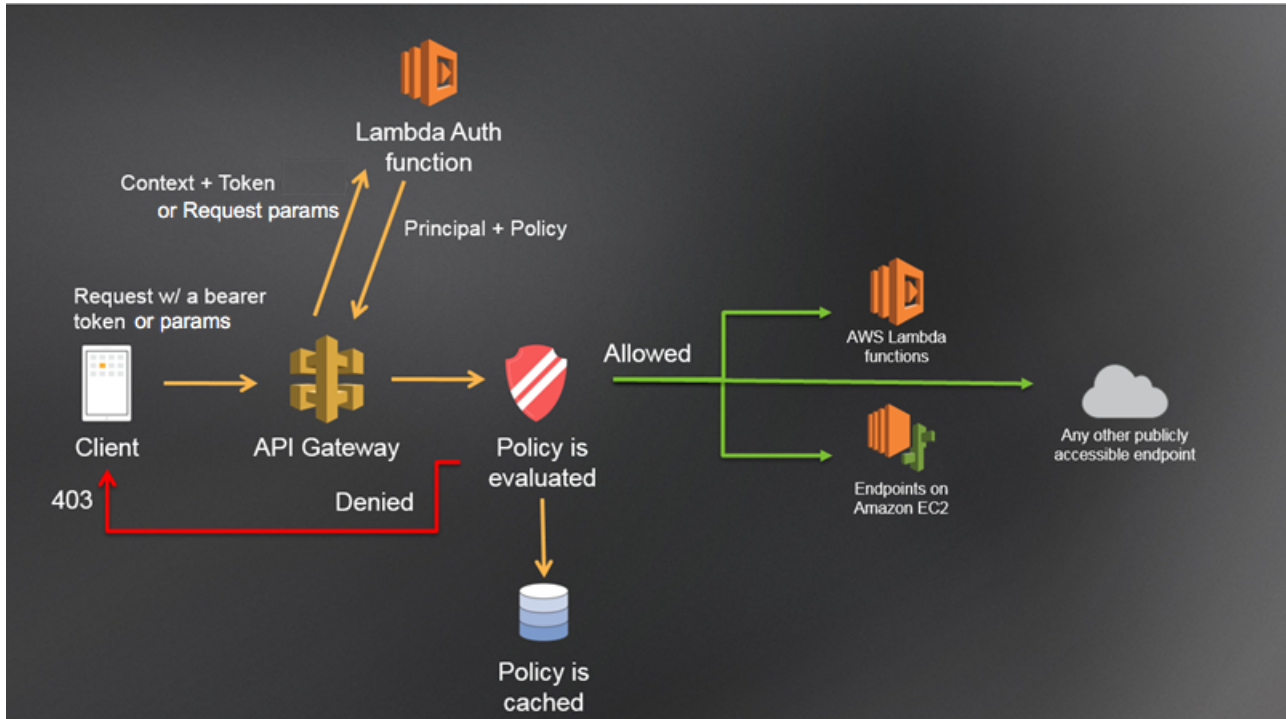


Fig. 11: Authorizer workflow

In this project the authorizer is used to protect the access to public endpoints. There are two different authorizer functions, one for admins and one for users. The workflow is the same, the only difference refers to the *client id* and the *public key* provided by *Auht0*, which are needed to authenticate the client.

2.5 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.

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;

```
service: serverless-user-management

plugins:
  - serverless-dotenv-plugin
  - serverless-plugin-split-stacks #to avoid the limit of 200 resources

custom:
  splitStacks:
    perFunction: false
    perType: true
  dotenv:
    include:
      - AUTHO_ADMIN_CLIENT_ID
      - AUTHO_ADMIN_CLIENT_PUBLIC_KEY
      - AUTHO_ADMIN_DOMAIN
      - AUTHO_USER_CLIENT_ID
      - AUTHO_USER_CLIENT_PUBLIC_KEY
      - AUTHO_USER_DOMAIN
```

Fig. 12: Plugins - serverless.yml

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

```
provider:
  name: aws
  runtime: nodejs10.x
  region: eu-central-1
  stage: dev
  iamRoleStatements:
    - Effect: "Allow"
      Resource: "*"
      Action:
        - "dynamodb:*"
        - "sqs:*"
        - "lambda:*"
        - "cloudwatch:*"
        - "apigateway:*
```

Fig. 13: Provider info - serverless.yml

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

```
UsersDynamoDBTable:
  Type: 'AWS::DynamoDB::Table'
  Properties:
    AttributeDefinitions:
      - AttributeName: userId
        AttributeType: S
    KeySchema:
      - AttributeName: userId
        KeyType: HASH
    ProvisionedThroughput:
      ReadCapacityUnits: 5
      WriteCapacityUnits: 5
    TableName: user
```

Fig. 14: 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;

```
commandUpdateUser:
  handler: handler.commandUpdateUser
  timeout: 10
  events:
    - sqs:
        arn: arn:aws:sqs:eu-central-1:582373673306:updateUserQueue
```

Fig. 15: 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.

```
readUser:
  handler: handler.readUser
  events:
    - http:
        path: /readUser
        method: get
        cors: true
        integration: lambda
        request:
          template:
            application/json: '{ "userId": "$input.params("userId")" }'
```

Fig. 16: 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.

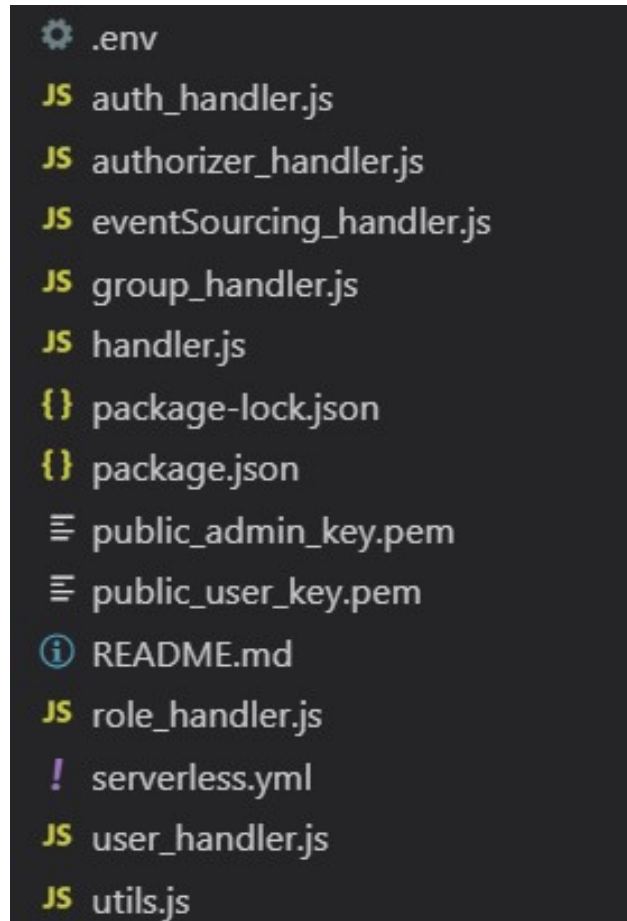


Fig. 17: Project's root

4 CQRS

Command Query Responsibility Segregation (CQRS) is an architectural pattern which separates the responsibility for modifying data (Command) from reading them (Query). The use of two different models for writing and reading operations, in scope of CQRS, allows instead to design and optimize each model for its responsibilities. In addition to this, the use of distinct models also allows the selection of the most appropriate technologies. As soon as the reading and writing models are separated, the infrastructure could easily scale to best fit the needs. It often happens that the number of writings in a system is much lower than the readings. Obviously the two models must be synchronized to ensure that the read information are consistent with the written ones.

The justification for CQRS is that in complex domains, a single model to handle both reads and writes gets too complicated, and we can simplify by separating the models.

The change that CQRS introduces is to split that conceptual model into separate models for update and display, which it refers to as Command and Query.

CQRS fits well with event-based programming models. It's common to see CQRS system split into separate services communicating with Event Collaboration. This allows these services to easily take advantage of Event Sourcing.

4.1 Architecture overview

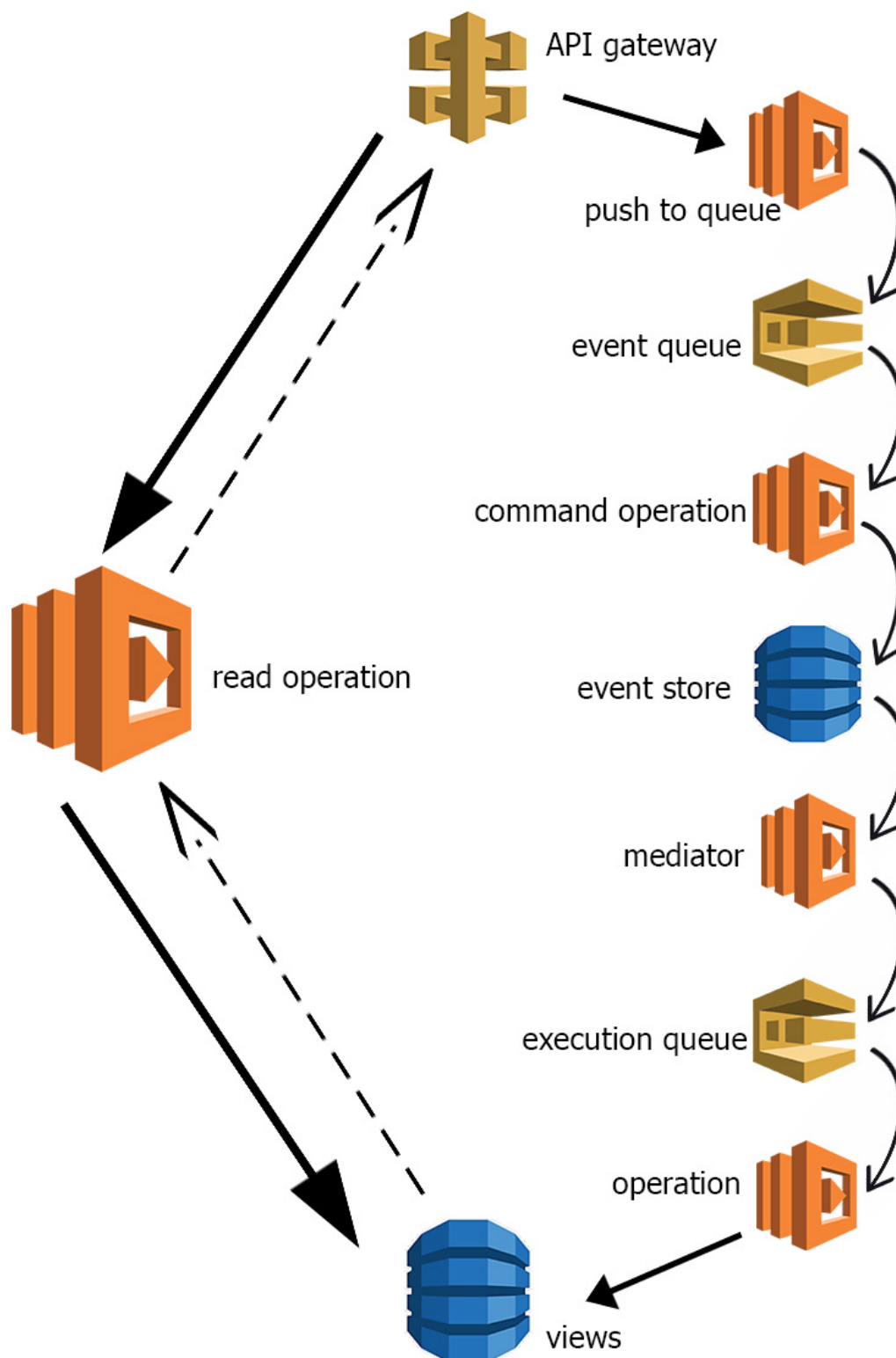


Fig. 18: Architecture overview

As you can see from the architecture overview, the application is separated into two models: write model(right side) and read model(left side).

The choice to apply the CQRS pattern due to separate the responsibilities of writing and reading, but also because read side events aren't stored into the event store table because they don't change the system's state but just retrieve information from it.

4.2 Write model

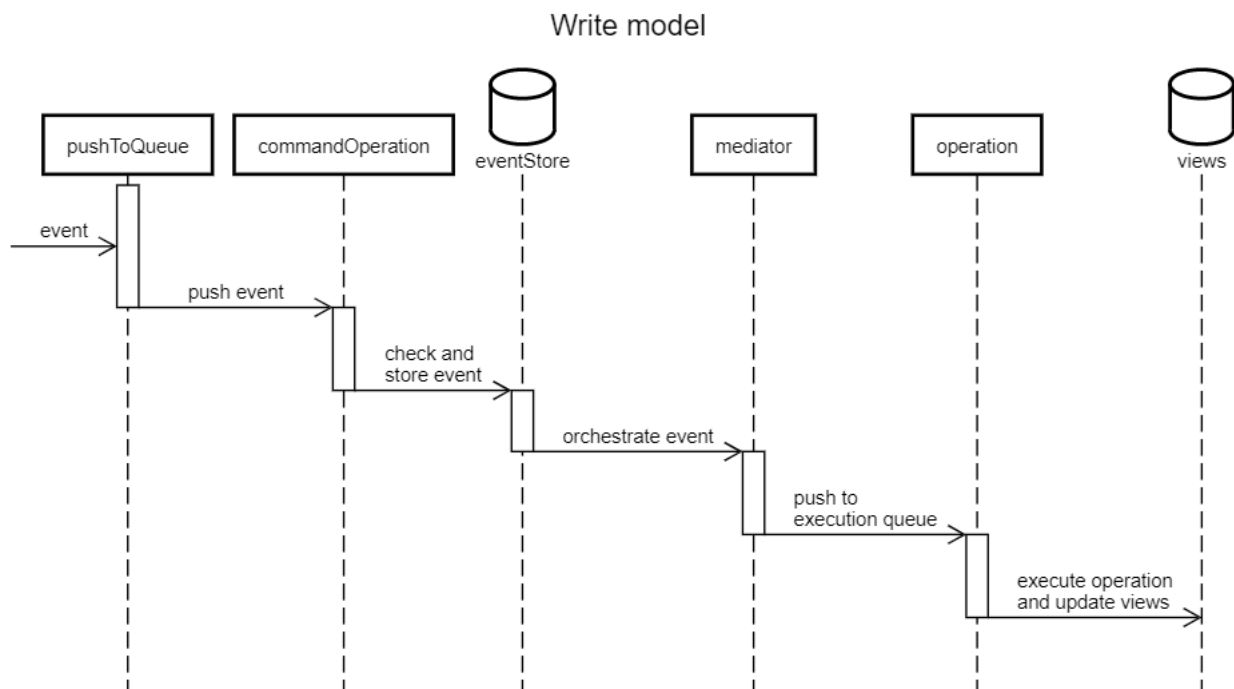


Fig. 19: Write model sequence diagram

4.3 Read model

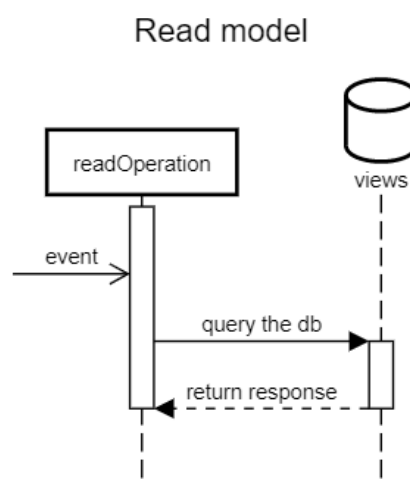


Fig. 20: Read model sequence diagram

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(without first 6 characters 'Auth0|');
- **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 is 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

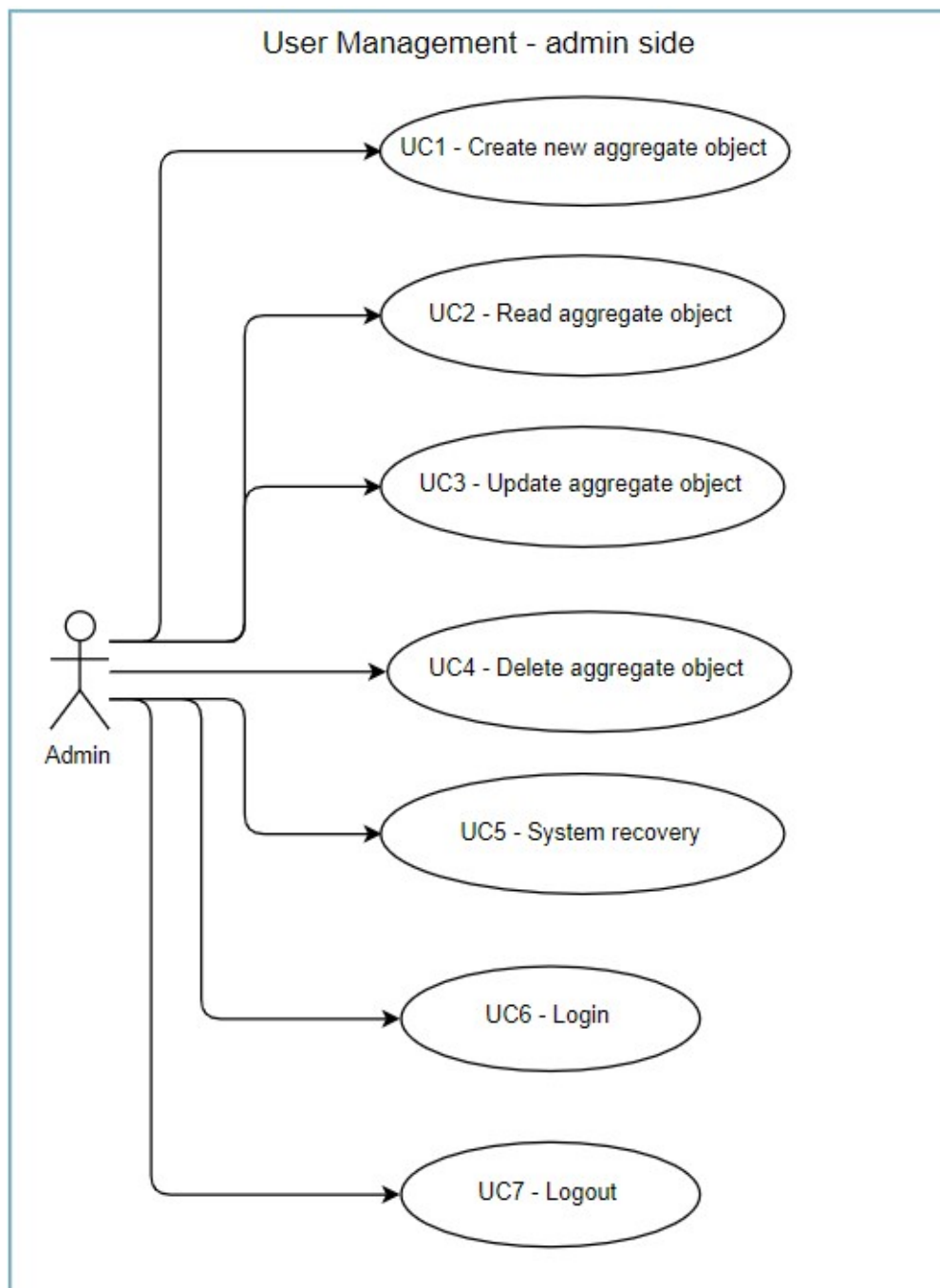


Fig. 21: Admin use cases

- UC1: the admin creates a new object based on the type of aggregate;
- UC2: the admin read the information about an object based on the type of aggregate;
- UC3: the admin updates an object based on the type of aggregate;
- UC4: the admin deletes an object;
- UC5: the admin recovers the system state from the chosen timestamp re-running of the event occurred after that;
- UC6: the admin log in into the application using the Auth0 portal;
- UC7: the admin log out from the application.

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

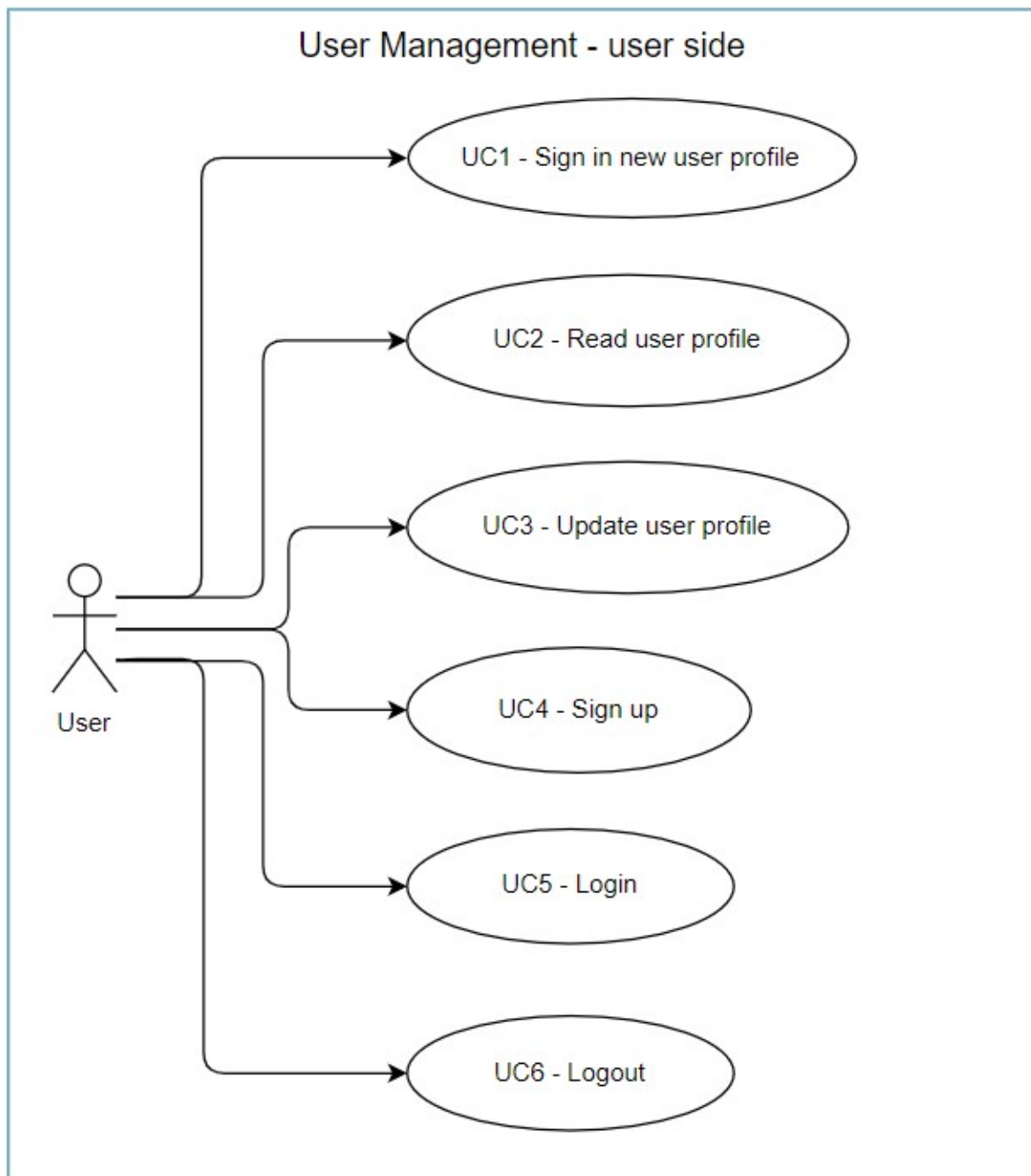


Fig. 22: User use cases

- UC1: the user fills the form to sign in into the application the first time logs in;
- UC2: the user read his profile;
- UC3: the user updates his profile;
- UC4: the user sign up to Auth0 authentication portal using email and password or Google's account;

- UC5: the user log in to Auth0 authentication portal using email and password or Google's account;
- UC6: the user log out from the application.

6 Event Sourcing

"Event Sourcing ensures that all changes to application are stored as a sequence of events."

You can query these events, use the event log to reconstruct past states and adjust the state to cope with retroactive changes.

The core idea of event sourcing is that whenever we make a change to the state of a system, we record that state change as an event, and we can confidently rebuild the system state by reprocessing the events at any time in the future.

When working with an event log can be useful to build snapshots of the working copy so that you don't have to process all the events when you rebuild the system or every time you need to query the database. For this reason when a new event is stored triggers another function to update the views.

Event sourcing can be used to:

- **Complete Rebuild:** you can discard the application state completely and rebuild it by re-running the events from the event store on an empty application.
- **Temporal Query:** you can determine the application state at any point in time. This can be used considering multiple time-lines (like branching in a VCS).
- **Event Replay:** if you find a past event was incorrect, you can replaying from then with the new event. The same technique can handle events received in the wrong sequence with systems that communicate with asynchronous messaging.

6.1 Event store

Event store is a database's table that contains all the events occurred from the begin. When using Event sourcing the event store becomes the principal source of truth and the system state is completely derived from it.

6.1.1 Event object

Each event stored in the table is a JSON object and respect this format:

- **EventId:** is a UUID;
- **Payload:** JSON object which contains all the information about the event;
- **Aggregate:** type of the aggregate that the event refers to;
- **ExecutionQueue:** name of the execution queue used to execute the event;
- **Timestamp:** number to know the execution order of the events.

```
aggregate String : user
eventId String : f7731a40-f634-354a-ac99-351074d83b6d
executionQueue String : executeCreateUserQueue
▶ payload Map {7}
timestamp Number : 1563543008991
```

Fig. 23: DynamoDB eventStore example item

7 Extension points

7.1 New aggregates

To create a new type of aggregate you have to:

- Define a new DynamoDB table in the *serverless.yml*;
- Create an *operationQueue* and an *executionQueue* with SQS;
- Define all the Lambda functions you need in the *serverless.yml*;
- Define the trigger events in the *serverless.yml*;
- Create a handler file named *"aggregate_handler.js"* in the project's root which contains the Lambda's handlers that refer the same aggregate type;
- Add the new handlers file in the *module.exports* of the main handler;
- Deploy the serverless application.

7.2 New write functions

To add a new write function you have to:

- Define the function's name and the corresponding handler's name in the *serverless.yml*;
- Define trigger events in the *serverless.yml*;
- Create an *operationAggregateQueue* and an *executeOperationAggregateQueue*;
- In the *"aggregate_handler.js"* you must write:
 - a *pushOperationAggregateToSQS* function to push the event in the corresponding queue;
 - a *commandOperationAggregate* function to check if the event is valid and store it into the *eventStore*;
 - an *operationAggregate* function to execute the event and update the view.
- Deploy the serverless application.

7.3 New read functions

To add a new read function you have to:

- Define the function's name and the corresponding handler's name in the *serverless.yml*;
- Define the corresponding get API endpoint in the *serverless.yml*;
- Write the *readOperation* function to query the database;
- Deploy the serverless application.

8 Setup

9 Upgrades