Building E-Commerce Order Management Systems - Technical Article #2 - Implementation

In a previous article, titled 'Building E-Commerce Order Management Systems - Technical Article #1 - Architecture', the TrackIt team outlined the different strategic and architectural choices developers can make to build robust, cost-effective, and flexible order management systems.

This second article in the E-Commerce Technical series focuses on the implementation process and details the different steps involved in setting up an OMS.

Step 1: Creating a Serverless Project Using Serverless Framework Documentation

The first step in the implementation process is to create a Serverless project. Readers can follow the instructions in the Serverless Framework Quick Start Guide to install the Serverless Framework open-source CLI and deploy a sample service that reports deployment information and other optional metrics to the Serverless Framework Dashboard.

Step 2: Setting Up the Logic for the OMS

The second step in the implementation process is to set up the logic for the order management system. This begins with the addition of the first step Function and Lambda functions.

Adding the First Step Function and Lambda Functions

We need to create the `serverless.yml` to define the basic Serverless configuration with Step Functions and Lambda functions. In the example below, the state machine has only one state `ProcessOrder:`

```
service: 'oms-article'

plugins:
  - serverless-step-functions

configValidationMode: error

provider:
  name: aws
```

```
  runtime: nodejs14.x
  memorySize: 1024 #
https://aws.amazon.com/fr/blogs/compute/operating-lambda-performance-opt
imization-part-2/
  region: 'us-west-2'

functions:
  processOrderHandler:
    handler: src/processOrder.handler

stepFunctions:
  stateMachines:
    processOrder:
      events:
        - http:
            path: processOrder
            method: GET
      definition:
        Comment: "Process Order - Process an order"
        StartAt: ProcessOrder
        States:
          ProcessOrder:
            Type: Task
            Resource:
              Fn::GetAtt: [processOrderHandler, Arn]
            End: true
```

Serverless Configuration

Next, we create the handler `src/processOrder/index.js`

```
const processOrderHandler = async () => {
  console.log('PROCESS ORDER HANDLER');
};

module.exports = {
  handler: processOrderHandler,
};
```

Handler Code Snippet

Configuring the OMS Database

After adding the first Step Function, the next step is to configure the OMS database and define a model that defines how data is stored within the database.

We use the Sequelize ORM (Object-Relational Mapping) to connect to our database - the configuration is added in `src/processOrder/index.js`. We also define a simple `Order` model

and create an entry every time the Step Function is called. The database table is created manually for the sake of simplicity.

```
const Sequelize = require('sequelize');

const dbConfig = {
  host: 'yourDBHost',
  port: 'yourDBPort',
  database: 'yourDBName',
  username: 'yourUsername',
  password: 'yourPassword',
  dialect: 'postgres', // one of 'mysql' | 'mariadb' | 'postgres' |
'mssql'
  schema: 'yourSchema',
  pool: { max: 5, idle: 30 },
  dialectOptions: {
    connect_timeout: 3000,
  },
};

const sequelize = new Sequelize(dbConfig.database, dbConfig.username,
dbConfig.password, dbConfig);

class Order extends Sequelize.Model {}
Order.init({
  id: {
    type: Sequelize.INTEGER,
    autoIncrement: true,
    primaryKey: true,
  },
  status: {
    type: Sequelize.STRING(20),
    allowNull: true,
  },
}, {
  sequelize,
  schema: sequelize.options.schema,
  tableName: 'order',
  createdAt: 'created_at',
  updatedAt: 'updated_at',
});

const processOrderHandler = async () => {
  console.log('PROCESS ORDER HANDLER');

  // We create a DB entry to save the order status
  const order = await Order.create({
    status: 'success',
  });
  return order;
};
```

```
module.exports = {
  handler: processOrderHandler,
  Order,
};
```

Sequelize Configuration and Model Definition

Sending Messages to the SQS Queues

Once the OMS database is configured and the orders are saved in the database for each execution, we set up an SQS queue that receives an order creation notification when a new order is created.

We attempt to send a message to a FIFO (First-In-First-Out) SQS queue - predominantly used to forward messages between services - and we verify that it was sent properly.

```
const Sequelize = require('sequelize');
const AWS = require('aws-sdk');

const dbConfig = {
 host: 'yourDBHost',
 port: 'yourDBPort',
 database: 'yourDBName',
 username: 'yourUsername',
 password: 'yourPassword',
 dialect: 'postgres', // one of 'mysql' | 'mariadb' | 'postgres' |
'mssql'
 schema: 'yourSchema',
 pool: { max: 5, idle: 30 },
 dialectOptions: {
   connect_timeout: 3000,
 },
};

const sequelize = new Sequelize(dbConfig.database, dbConfig.username,
dbConfig.password, dbConfig);

class Order extends Sequelize.Model {}
Order.init({
 id: {
   type: Sequelize.INTEGER,
   autoIncrement: true,
   primaryKey: true,
 },
 status: {
   type: Sequelize.STRING(20),
   allowNull: true,
 },
}, {
 sequelize,
 schema: sequelize.options.schema,
```

```
 tableName: 'order',
 createdAt: 'created_at',
 updatedAt: 'updated_at',
});

const sqs = new AWS.SQS({ apiVersion: '2020-10-06' });

const processOrderHandler = async () => {
 console.log('PROCESS ORDER HANDLER');

 // We create a DB entry to save the order status
 const order = await Order.create({
   status: 'success',
 });

 // We attempt to send a message to the queue
 const res = await sqs.sendMessage({
   MessageBody: JSON.stringify({
     message: 'the order has been processed',
     id: order.id,
   }),
   QueueUrl: 'https://yourQueueUrl',
   // MessageGroupId and MessageDeduplicationId params only apply to
FIFO queues, can be removed otherwise
   MessageGroupId: 'yourGroupId',
   MessageDeduplicationId: Math.floor(Math.random() *
Math.floor(1000000)).toString(),
 }).promise();
 if (res.stack) {
   console.log('Error while sending message to queue', { res });
 }
 return order;
};

module.exports = {
 handler: processOrderHandler,
 Order,
};
```

SQS Queue Configuration + Message Sending and Verification

Adding Error Handling

Adding error handling to automatically re-execute Lambdas in case of errors, such as temporary database connection issues, is important when setting up a reliable and robust order management system.

We edit the `serverless.yml` file to add a retry mechanism. The interval, max attempts and backoff rate need to be adapted to the reader's project.

```
service: 'oms-article'

plugins:
  - serverless-step-functions

configValidationMode: error

provider:
  name: aws
  runtime: nodejs14.x
  memorySize: 1024 # in MB. vCPUs are proportional to the memory size.
1024MB is often the most cost effective allocation
  region: 'us-west-2'

functions:
  processOrderHandler:
    handler: src/processOrder.handler

stepFunctions:
  stateMachines:
    processOrder:
      events:
        - http:
            path: processOrder
            method: GET
      definition:
        Comment: "Process Order - Process an order"
        StartAt: ProcessOrder
        States:
          ProcessOrder:
            Type: Task
            Resource:
              Fn::GetAtt: [processOrderHandler, Arn]
            End: true
            Retry:
                - ErrorEquals:
                  - States.ALL
                  IntervalSeconds: 5
                  MaxAttempts: 5
                  BackoffRate: 2
```

Adding the Retry Mechanism to the Serverless Configuration File

Adding GraphQL to Expose OMS Data to Other Services

Adding GraphQL is essential when users wish to implement external services that request OMS data for their functioning.

The `serverless.yml` file is edited to add the AppSync configuration that enables users to configure the GraphQL API and make it available to third-party services.

```
service: 'oms-article'

plugins:
  - serverless-step-functions
  - serverless-appsync-plugin

configValidationMode: error

provider:
  name: aws
  runtime: nodejs14.x
  memorySize: 1024 # in MB. vCPUs are proportional to the memory size.
1024MB is often the most cost effective allocation
  region: 'us-west-2'

functions:
  processOrderHandler:
    handler: src/processOrder.handler
  orderResolver:
    handler: appSync/resolvers/OrderResolver.handler

stepFunctions:
  stateMachines:
    processOrder:
      events:
        - http:
            path: processOrder
            method: GET
      definition:
        Comment: "Process Order - Process an order"
        StartAt: ProcessOrder
        States:
          ProcessOrder:
            Type: Task
            Resource:
              Fn::GetAtt: [processOrderHandler, Arn]
            End: true
            Retry:
                - ErrorEquals:
                  - States.ALL
                  IntervalSeconds: 5
                  MaxAttempts: 5
                  BackoffRate: 2

custom:
  appSync:
    name: OMS-dev
    authenticationType: AWS_IAM
    schema: "./appSync/schemas/schema.graphql"
    dataSources:
      - type: AWS_LAMBDA
        name: OrderResolver
        description: 'order resolver'
```

```
        config:
          functionName: orderResolver
    defaultMappingTemplates: # default templates. Useful for Lambda
templates that are often repetitive. Will be used if the template is not
specified in a resolver
      request: "./requests/default.vtl"
      response: "./responses/default.vtl"
    mappingTemplatesLocation: "./appSync/mappingTemplates"
    mappingTemplates:
      - dataSource: OrderResolver
        type: Query
        field: getOrder
```

Adding the AppSync Configuration to the Serverless Configuration File

The next step is to create a GraphQl schema (`appSync/schemas/schema.graphql`) that matches the order table:

```
schema {
    query: Query
}

type Query {
    getOrder(id: ID!): Order
}

type Order {
    id: ID!
    status: String
}
```

Creating a GraphQL Schema for the Order table

We then need to create default mapping templates to define the request/response format for the handler.

- `appSync/mappingTemplates/requests/default.vtl`

```
#**
The value of 'payload' after the template has been evaluated
will be passed as the event to AWS Lambda.
*#
{
  "version" : "2017-02-28",
  "operation": "Invoke",
  "payload": {
    "params": $utils.toJson($ctx.args),
```

```
    "selectionSetList": $utils.toJson($ctx.info.selectionSetList)
  }
}
```

Request Mapping Template

- `appSync/mappingTemplates/responses/default.vtl`

```
$util.toJson($context.result)
```

Response Mapping Template

Lastly, we define the GraphQL resolver `appSync/resolvers/OrderResolver.js` that processes the requests and prepares the response.

```
const {
  Order,
} = require('../../src/processOrder');

const fieldsMapping = {
  id: {
    request: (request) => ({
      ...request,
      attributes: [
        ...(request.attributes || []),
        'id',
      ],
    }),
    response: (order, response) => ({
      ...response,
      id: order.id,
    }),
  },
  status: {
    request: (request) => ({
      ...request,
      attributes: [
        ...(request.attributes || []),
        'status',
      ],
    }),
    response: (order, response) => ({
      ...response,
      status: order.status,
    }),
  },
};

const buildRequest = ({ params, selectionSetList }) => {
```

```
  const baseRequest = {
    where: {
      id: params.id,
    },
  };
  return selectionSetList.reduce((request, field) =>
fieldsMapping[field].request(request), baseRequest);
};

const buildResponse = ({ data, selectionSetList }) => {
  const baseResponse = {};

  return selectionSetList.reduce(
    (response, field) => fieldsMapping[field].response(data, response),
    baseResponse,
  );
};

const handler = async (event) => {
  console.log('ORDER RESOLVER HANDLER');
  const request = buildRequest({ params: event.params, selectionSetList:
event.selectionSetList });
  const data = await Order.findOne(request);
  const response = buildResponse({ data, selectionSetList:
event.selectionSetList });
  return response;
};

module.exports = {
  handler,
};
```

Adding the GraphQL Resolver

We can now execute GraphQL queries on the endpoint or on the AWS console. Here is an example:

```
query MyQuery {
  getOrder(id: "4") {
    id
    status
  }
}
```

Example of a GraphQL Query

Additional Information

The `package.json` file used for this project is as follows:

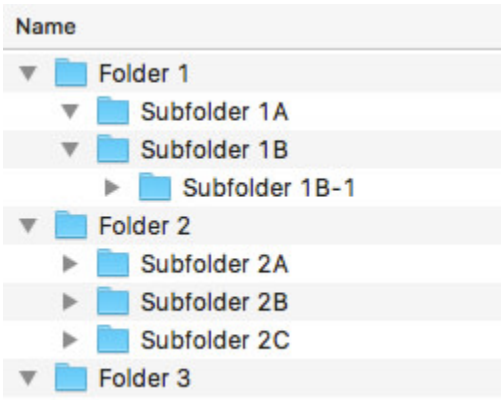
```
{
  "name": "oms-article",
  "version": "1.0.0",
  "description": "oms-article",
  "main": "index.js",
  "dependencies": {
    "aws-sdk": "^2.793.0",
    "sequelize": "^5.22.3",
    "pg": "^8.2.1"
  },
  "devDependencies": {
    "eslint": "^7.1.0",
    "eslint-config-airbnb-base": "^14.1.0",
    "eslint-plugin-import": "^2.20.2",
    "sequelize-cli": "^5.5.1",
    "serverless": "^2.41.2",
    "serverless-appsync-plugin": "^1.11.3",
    "serverless-step-functions": "^2.30.0"
  },
  "author": "",
  "license": ""
}
```

The package.json File Used in this Article

At this point, readers will have implemented a Serverless project that's connected to an OMS database and also has the basic logic required to run an order management system.

Step 3: Resolving Common Issues

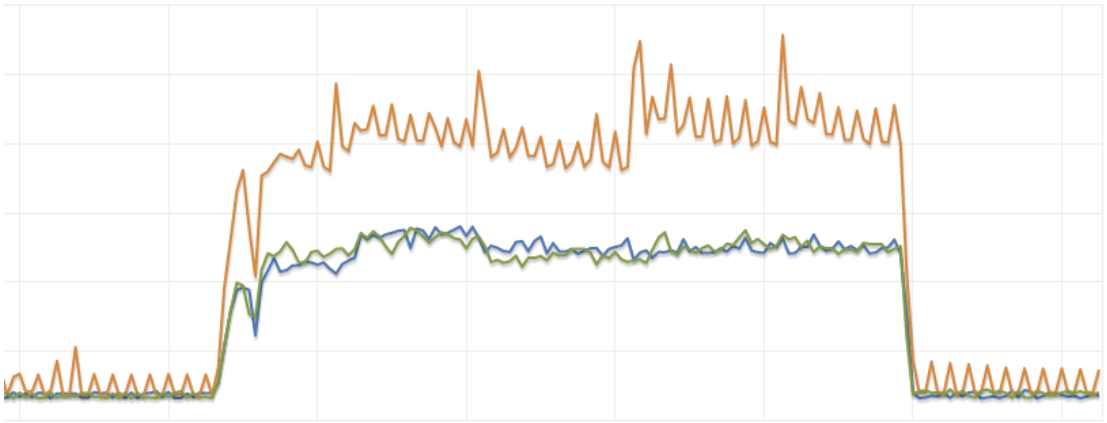
Common Issue #1: Organizing the Project when there are many Lambdas and Step Functions



A Well-Organized Project

Solution: Developers can split the project into different folders to ensure readability.

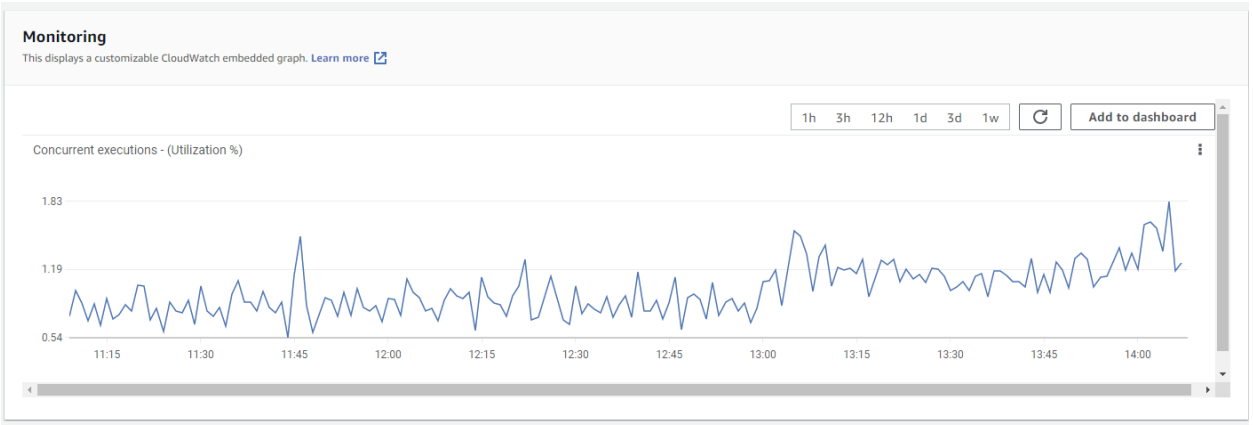
Common Issue #2: Excessive Database Load



Database Load

Solution: Developers can create SQL indexes to optimize SELECT data requests.

Common Issue #3: Lambda Quota/Rate Limit



Concurrent Lambda Executions

Solution: Companies can Contact AWS to request higher quotas on Lambda functions.

Common Issue #4: Setting up an Environment for Dev Testing

To develop features, developers should have their own environment to run tests. Developers often realize that it's not easy to run E2E tests since they require a lot of setup.

Solution: Developers can write a script that creates the database, deploys the stack, generates the env file, and sets up all the default values and migrations.

```
const { generateDatabase } = require('./databaseGenerator');
const { generateServerless } = require('./serverlessGenerator');
```

```
const { generateEnv } = require('./envGenerator');
const { questionUser } = require('./questionUser');
const { generateQueues } = require('./queuesGenerator');

const config = {
  SERVICE_NAME: 'oms',
  REGION: 'us-west-2',
  NODE_ENV: 'test_qa',
  API_GATEWAY_ENDPOINT_TYPE: 'regional',
};

// UNIT TESTS
const unitEnv = {
  TEST_HOST: '127.0.0.1',
  TEST_PORT: '5432',
  TEST_DB: 'postgres',
  TEST_USER: 'postgres',
  TEST_PASSWORD: 'postgres',
};

const main = async () => {
  const { answersQa, envName } = await questionUser();
  const queuesEnv = await generateQueues(answersQa.STAGE_QA);
  const context = {
    name: envName, config, answersQa, queuesEnv, unitEnv,
  };
  await generateDatabase(answersQa);
  await generateEnv(context);
  const endpoint = await generateServerless(context);
  await generateEnv({
    ...context, endpoint,
  });
};

try {
  main()
    .then((r) => r)
    .catch((err) => console.log(err));
} catch (err) {
  console.log(err);
}
```

Script to Generate Dev Environment for Testing

Conclusion

This article has detailed the steps readers can take to implement robust order management systems and has also addressed some of the most common issues developers face when setting up an OMS. The next article in the E-Commerce Technical series will focus on OMS

testing and different considerations companies need to make in order to test and ensure the proper functioning of their order management systems.

About TrackIt

TrackIt is an Amazon Web Services Advanced Consulting Partner specializing in cloud management, consulting, and software development solutions based in Venice, CA.

TrackIt specializes in Modern Software Development, DevOps, Infrastructure-As-Code, Serverless, CI/CD, and Containerization with specialized expertise in Media & Entertainment workflows, High-Performance Computing environments, and data storage.

TrackIt's forté is cutting-edge software design with deep expertise in containerization, serverless architectures, and innovative pipeline development. The TrackIt team can help you architect, design, build, and deploy customized solutions tailored to your exact requirements.

In addition to providing cloud management, consulting, and modern software development services, TrackIt also provides an open-source AWS cost management tool that allows users to optimize their costs and resources on AWS.