Serverless Application on AWS

Project report submitted in partial fulfillment of the requirement for the degree of Bachelor of Technology

in

Computer Science and Engineering/Information Technology

By

Aditya Sharma (181395)

Under the supervision of

Dr. Rajinder Sandhu

To

Department of Computer Science & Engineering and Information Technology **Jaypee University of Information Technology Waknaghat, Solan-173234,**

Himachal Pradesh

CERTIFICATE

Candidate's Declaration

I hereby declare that the work presented in this report entitled **"Serverless application on AWS"** in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering/Information Technology** submitted in the department of Computer Science & Engineering and Information Technology**,** Jaypee University of Information Technology Waknaghat is an authentic record of my own work carried out over a period from February 2022 to July 2022 under the supervision of **Dr. Rajinder Sandhu**

(Assistant professor Senior grade Computer Science Engineering).

The matter embodied in the report has not been submitted for the award of any other degree or diploma.

Aditya Sharma-181395

This is to certify that the above statement made by the candidate is true to the best of my knowledge.

Dr. Rajinder Sandhu

Assistant Professor (SG)

Computer Science Engineering Dated: 01/12/2021

ACKNOWLEDGEMENT

To begin with, I want to express my heartfelt gratitude to almighty God for His divine blessing, which has enabled us to successfully complete the project work.

Supervisor Dr.Rajinder Sandhu, Assistant Professor (SG), Department of CSE Jaypee University of Information Technology, Waknaghat, deserves my deepest gratitude. My supervisor's deep knowledge and keen interest in the field of "Data Science and Machine Learning" made it possible to complete this project, as well as his relentless patience, scholarly guidance, persistent and enthusiastic supervision, constructive criticism, insightful advice, and reading many inferior draughts and correcting them at all stages.

I'd like to express my heartfelt gratitude to Dr. Rajinder Sandhu of the Department of CSE for his invaluable assistance in completing my thesis.

I'd also like to express my gratitude to everyone who has assisted me in making this project a success, whether directly or indirectly. In this specific circumstance, I'd like to express my gratitude to the numerous staff members, both teaching and non-teaching, who have provided me with valuable assistance and facilitated my project.

Finally, I must express my gratitude for my parents' unwavering and unflagging love, support and utmost patience in putting up with me.

Aditya Sharma(181395)

ABOUT THE COMPANY

Andris Zoltners and Prabhakant (Prabha) Sinha, two Ph.D. classmates who eventually became college professors at Northwestern University's Kellogg School of Management, created ZS in 1983.

ZS assisted eight of the top ten pharmaceutical corporations in the world align regions and shrink their sales forces in its first three years. By 2011, ZS has partnered with 49 of the top 50 pharmaceutical companies and 17 of the top 20 medical device companies.

Today, ZS works with firms all around the world and in a variety of areas, including healthcare, high-tech, financial services, and more. ZS's knowledge and services have also grown dramatically. We now assist clients with everything from research to commercialization, as well as the strategy, analytics, and technology required to make it all possible.

ABSTRACT

Serverless computing is a relatively new cloud programming and deployment paradigm that is gaining a lot of traction. Serverless products such as Amazon Web Services (AWS) Lambda, Google Functions, and Azure Functions automatically execute simple functions uploaded by developers, in response to cloud-based event triggers. The serverless abstraction makes integrating concurrency and parallelism into cloud applications much easier, and it allows for the low-cost deployment of scalable distributed systems and services.

In this work we have used an API gateway in AWS by making use of a SAM template for information extraction and modification. The gateway is triggered by a lambda function written in Python and this data is stored in a database in a sequential ordering by making use of SNS Queue. Another layer of security is added by making use of Amazon Cognito for Authentication purposes.

TABLE OF CONTENTS

CHAPTER 01: INTRODUCTION

1.1 PROJECT INTRODUCTION

The AWS Serverless Application Model (SAM) denotes a framework that consists of an open source template to build serverless applications. It provides shorthand syntax to express functions, APIs, databases, and event source mappings. You can define the application you want and model it using YAML with just a few lines per resource. SAM transforms and expands the SAM syntax into AWS CloudFormation syntax during deployment, enabling you to build serverless applications faster.

The AWS SAM CLI is required to begin developing SAM-based applications. SAM CLI is a Lambda-like execution environment that allows you to build, test, and debug apps locally using SAM templates or the AWS Cloud Development Kit (CDK). The SAM CLI can be used to deploy applications to AWS or to build secure continuous integration and deployment (CI/CD) pipelines that interact with AWS' native and third-party CI/CD systems.

Shorthand syntax for functions, APIs, databases, and event source mappings is provided by AWS SAM. SAM converts and expands SAM syntax into AWS CloudFormation syntax during deployment. After that, CloudFormation provides a very reliable deployment tool for the resources employed.

A serverless application's architecture is represented by a SAM template file, which is a YAML configuration. The template is used to declare all of the AWS resources that make up your serverless app in one place. Any resource that you may declare in an AWS CloudFormation template can likewise be defined in an AWS SAM template, making it easy to work with.

1.2 MOTIVATION

India is the home to the one of the largest consumer markets in the world. During the financial year 2021-2022, the volume of digital payments in India increased by 33% year on year . According to the Ministry of Electronics and Information Technology, there were 7,422 crore digital payment transactions in FY 2020-21, up from 5,554 crore in FY 2020-21. Order processing is used many phases of large scale businesses eg. payment business, delivery business etc. Digital transactions are becoming increasingly widespread in everyday life. From 2020 to 2021, transactions connected to telecom and electricity bills increased by 3640 percent and 2353 percent, respectively. In 2020, lifestyle and fashion were the leading contributors to e-commerce transactions, but in 2021, groceries overtook them with a 233 percent increase. Fitness-related transactions increased by 611 percent in 2021, indicating that more individuals are becoming health-conscious. During the holiday season, nearly one-third of all e-commerce transactions occurred, as projected. With remote and hybrid work becoming the norm, professionals may be settling back in their hometowns, since this sector grew by over 210 percent in tier 2 and tier 3 cities between 2010 and 2020. Online meal orders have increased in popularity by 284.89 percent in the last year. While consumer-driven e-commerce dominated in 2020, Wholesale E-commerce expanded by over 1500% in 2021, indicating a rise in business digital adoption. The popularity of Fantasy League and Esports has led to an increase in usage in the Games sector. Game developer tools were in demand due to increased consumer demand for games, with transaction volume for the sub-sector increasing by 365.83 percent in 2021. To manage such a huge load of data the system that needs to be built must have sufficient fault tolerance to process such transactions. The API provides a level of abstraction that can help to extract only the data which can be revealed without compromising confidential data. Such systems provide a way for smooth communication between client and the server to ensure smooth transactions. Therefore AWS is a scalable platform wherein a Serverless system can be deployed for the purpose of managing the transactions received by building fault tolerant and robust systems. AWS SAM makes it simple to organize and manage related components and resources on a single stack. AWS SAM allows you to share resource configuration (such as RAM and timeouts) and deploy all associated resources as a single, versioned entity.

1.3 OBJECTIVES

- Make use of SAM AWS template due to the various advantages offered
- Build an API gateway by using SAM template and storing the configurations in template.yml file
- Initialize Dynamo DB database for storing data that will be extracted
- Creating lambda function to trigger the API gateway to perform various operations.
- Initialize a SNS queue to perform order processing and store it in the way they come in
- Use Authentication method to provide a layer of security for the system.

1.4 TOOLS USED

This project makes use of the Python programming language for its implementation. The use of Python was required for creating lambda functions to use the API gateway created. The AWS services employed during the project are listed below:

- AWS Lambda
- Amazon API Gateway
- Amazon DynamoDB
- AWS Step Functions
- Amazon Cognito
- Amazon cloud formation

CHAPTER 02: LITERATURE SURVEY

The Serverless framework[1] streamlines the Lambda application development process. Using an offline mode, users can debug Lambda functions locally with this plugin. Dockerlambda is a reverse-engineered sandbox for AWS Lambda. It is compatible with all Lambda runtimes. On AWS Lambda, this guarantees the same behaviour. Josef Spillner researched FaaS and put Snafu[2] into practising a modular framework that works with AWS LambdaDebug Lambda apps with this tool.

The only other existing AWS debugging and analysis support CloudWatch and X-Ray, two logging tools, are used to create Lambda applications. AWS Lambda monitoring is provided by New Relic[3] and Dashbird[4], which summarise data. Courtesy of AWS Cloudwatch Zipkin[5] is a Google Dapper[35]-based distributed tracing system. Obtaining causal ordering to aid debugging Performance analysis is well-known and widely used has been thoroughly investigated.

According to Schwarz et al.[6], understanding distributed applications requires describing the causal link. Bailis et al. review causality in [7] in the context of real-world applications and propose a number of intriguing model expansions.

In distributed applications, several systems track causal links. Dapper[8], Google's production distributed systems tracing infrastructure, is now open to the public. The report explains how they achieved reduced overhead, application-level transparency, and large-scale deployment.

Fonseca et al. presented X-trace[9], a tracing framework that provides a comprehensive view of behaviour of a modern Internet system with many applications distributed across various administrative domains. Kronos [10] determines the order of interdependent processes in a distributed system via a separate event ordering service system. Several sample applications are used by Escriva et al. to highlight the value of having a Kronos API.

New ways to achieving causal consistency in distributed and scalable datastore systems are being developed in other studies. COPS[11], a key-value store suggested by Lloyd et al., provides causal consistency across a large area. They discover and define a new consistency model known as causal consistency with convergent conflict resolution. Bolt-on[12] is a solution suggested by Bailis et al. that provides a shim layer on top of general-purpose and widely deployed datastores that provides causal consistency.

Saturn[13] is a geo-replicated data management system metadata service. It can be used to ensure that remote activities are carried out in a visible and causally correct order. Saturn has been tested on Amazon EC2, and the results show that weakly consistent datastores can outperform finally consistent models (by causal consistency).

CHAPTER 03: MAJOR PROJECT SDLC

3.1 Feasibility Study

3.1.1 Financial Feasibility

Our project does not take up any major financial investment, but the cost for AWS account for services was incurred. The baseline costs involved are mentioned below:

- DynamoDB: <$0.1
- API Gateway: <$0.1
- Step Functions: <$0.1
- S3: <$0.1
- Cognito: <$0.1
- SQS: <$0.1

3.1.2 Technical Feasibility

The project resulted in Serverless application at Scale on AWS platform can be used to make order requests by making use of API gateway. This scalable application can take into account the workload incurred and ramp up or decrease the resources needed accordingly. The Serverless Application Management template allows easy communication between client and server by means of lambda function which can be programmed in any language desired by the user (e.g. python, javascript, etc.) by making use of frameworks like Django or node.js etc. The project required creating an AWS account and setting up a Cloud9 workshop with required services employed for its completion.

3.2 Use Case Diagram

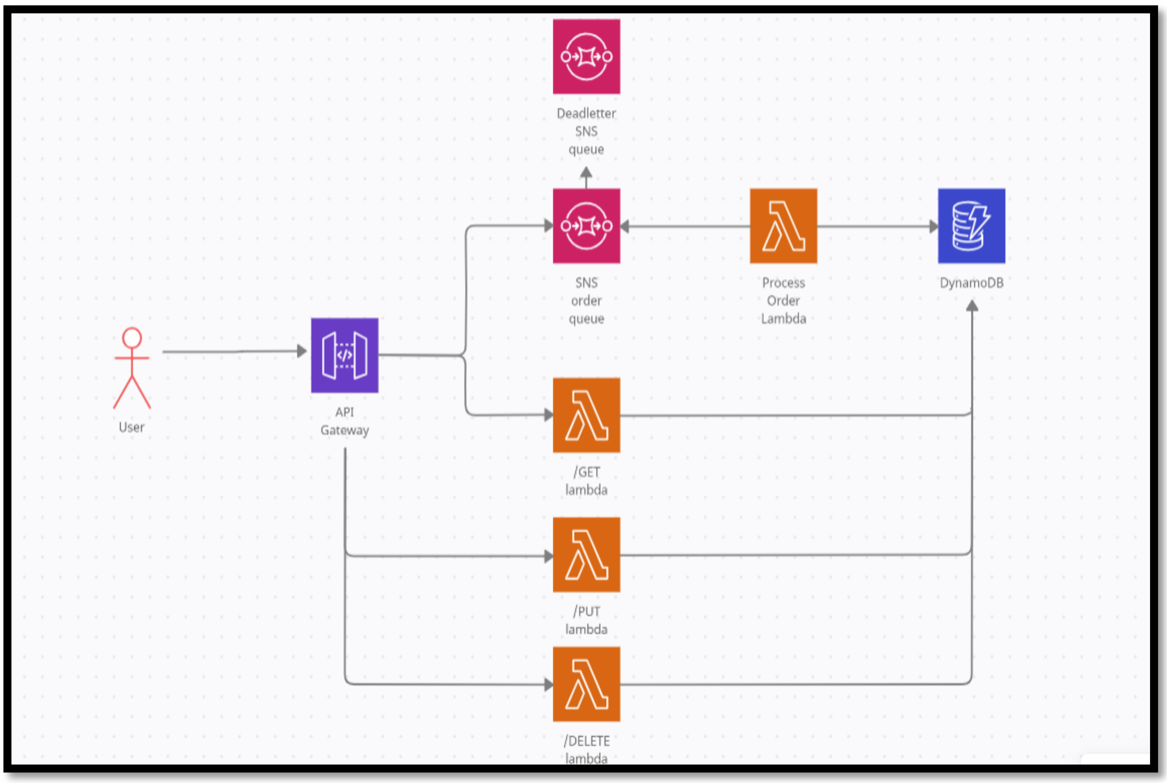


Fig 1: Use Case Diagram

3.3 DF Diagram

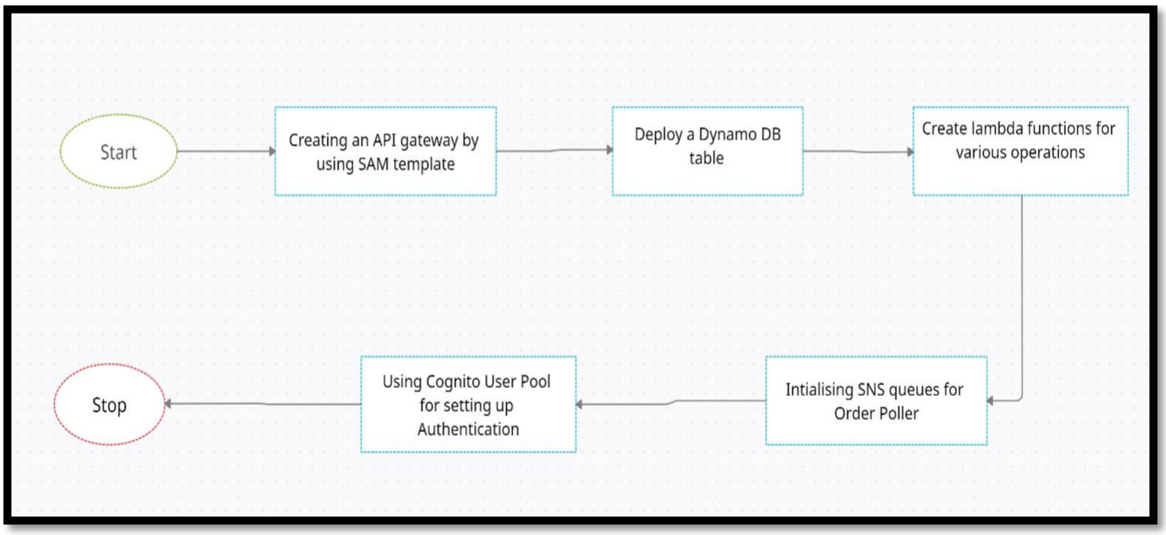

Start
Creating an API gateway by using SAM template
Deploy a Dynamo DB table
Create lambda functions for various operations
Intialising SNS queues for Order Poller
Using Cognito User Pool for setting up Authentication
Stop

Fig 2: DF Diagram

3.4 AWS Services used

AWS Lambda

AWS Lambda is a serverless compute service that enables you to run code without having to provision or manage servers, write workload-aware cluster scaling logic, maintain event integrations, or manage runtimes. You can run code for nearly any form of application or backend service with Lambda, and you don't have to worry about administration. Simply upload your code as a ZIP file or container image, and Lambda will assign compute execution power and run your code based on the incoming request or event, at any scale. It can be used for over 200 AWS services and SaaS applications to trigger the code, or can call it directly from any web or mobile app . This helps in easy connection with API gateway endpoint to send request from the client
Lambda functions may be used to develop in whatever language you choose (Node.js, Python, Go, Java, and more) and build, test, and deploy them using both serverless and container tools like AWS SAM or Docker CLI.

Amazon API Gateway

The Amazon API Gateway service is a fully managed service that allows developers to easily construct, publish, maintain, monitor, and protect APIs at any size. APIs allow applications to access data, business logic, and functionality from your backend services through a "front door." RESTful APIs and WebSocket APIs can be created with API Gateway to enable real-time two-way communication applications. Web applications, as well as containerized and serverless workloads, are supported by API Gateway.
API Gateway takes care of everything from traffic management to CORS support to authorization and access control to throttling, monitoring, and API version management for hundreds of thousands of concurrent API calls. There are no minimum fees or initial costs with API Gateway. You pay for the API calls you receive and the data you send out, and you can save money by using the API Gateway tiered pricing model as your API usage grows.

Amazon Simple Queue Service (SQS)

SQS is a fully managed message queuing service from Amazon that lets you decouple and scale microservices, distributed systems, and serverless applications. SQS removes the complexity and overhead of managing and operating message-oriented middleware, allowing developers to concentrate on work that is unique. You can send, store, and receive messages across software components using SQS at any volume without losing messages or necessitating the availability of other services. Using the AWS dashboard, your preferred Command Line Interface or SDK, and three easy commands, you can get started with SQS in minutes.

SQS provides two different types of message queues. Standard queues provide high throughput, best-effort ordering, and delivery at least once. SQS FIFO queues are meant to ensure that messages are processed only once, in the sequence in which they are received.

Amazon DynamoDB

At any scale, Amazon DynamoDB is a key-value and document database with single-digit millisecond performance. It's a fully managed, multi-region, multi-active, persistent database for internet-scale applications with built-in security, backup and restore, and in-memory caching. DynamoDB can handle over 10 trillion requests per day and up to 20 million requests per second at its peak.

AWS Step Functions

AWS Step Functions is a low-code visual workflow utility for orchestrating AWS services, automating business processes, and developing serverless apps. Workflows take care of failures, retries, parallelization, service integrations, and observability, allowing developers to concentrate on higher-value business logic.

Amazon Cognito

Amazon Cognito allows you to quickly and easily add user sign-up, sign-in, and access management to your online and mobile apps. Amazon Cognito scales to millions of people and supports sign-in via SAML 2.0 and OpenID Connect with social identity providers like Apple, Facebook, Google, and Amazon, as well as enterprise identity providers.

Amazon cloud formation

AWS CloudFormation is a service that assists you in modelling and configuring your AWS resources so you can spend less time maintaining them and more time working on your AWS-based apps. You construct a template that outlines all of the AWS resources you want (such as Amazon EC2 instances or Amazon RDS DB instances), and CloudFormation handles provisioning and configuration for you. CloudFormation handles the creation and configuration of AWS resources as well as determining what is dependent on what. The situations below show how CloudFormation can assist.

AWS CloudFormation performs underlying service calls to AWS to provision and configure your resources when you create a stack. Only activities that you have authority to conduct can be performed by CloudFormation. For example, permissions to create instances are required to use CloudFormation to generate EC2 instances. When you delete stacks with instances, you'll need equivalent permissions to terminate instances. To handle permissions, you utilise AWS Identity and Access Management (IAM).

Your template declares all of the calls that CloudFormation does. Take, for example, a template that describes an EC2 instance with the t2.micro instance type. When you use that template to build a stack, CloudFormation uses the Amazon EC2 create instance API with the t2.micro instance type.

3.5 Authentication Methods

Lambda Authorizers

A Lambda authorizer (previously known as a custom authorizer) is a Lambda function that controls API access. This Lambda function is invoked using a request context or an authorization token provided by the client application when your API is called. If the caller is authorised to do the specified operation, the Lambda function returns true.

IAM Permissions

AWS Identity and Access Management (IAM) permissions let you manage who can use your API. IAM credentials must be used to authenticate users calling your API. Only an IAM policy associated to the IAM user that represents the API caller, an IAM group that contains the user, or an IAM role that the user assumes will allow calls to your API to succeed.

Amazon Cognito user pools

Amazon Cognito user pools are Amazon Cognito user directories. A client of your API must first sign a user into the user pool and get the user's identity or access token. The client then uses one of the returned tokens to call your API. Only if the needed token is valid does the API call succeed.

API Keys –

API keys are alphanumeric string values that you give to app developers to gain access to your API.

Resource Policies

JSON policy documents that you can connect to an API Gateway API are known as resource policies. To control whether a specific principal (usually an IAM user or role) can invoke the API, utilise resource policies.

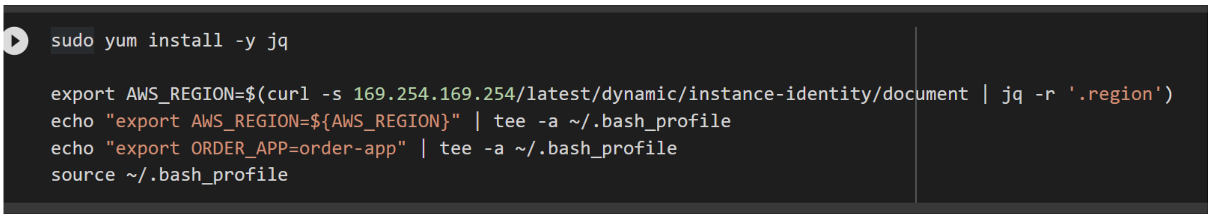
OAuth 2.0/JWT authorizers

To limit access to your APIs, you can use JWTs as part of the OpenID Connect (OIDC) and OAuth 2.0 frameworks. API Gateway validates JWTs that clients submit with API queries and permits or refuses requests depending on token validation and, optionally, token scopes.

CHAPTER 04: IMPLEMENTATION

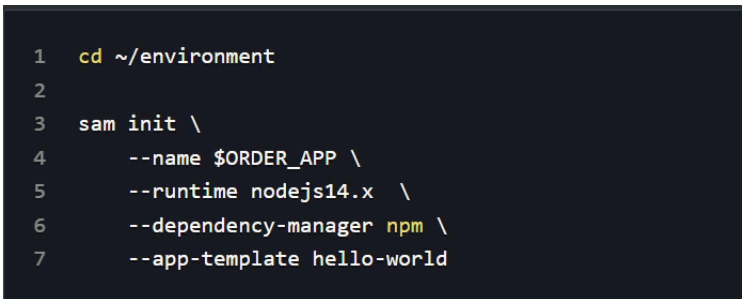
Setting up the environment

First of all we install jq a lightweight package for processing Json files and set the AWS region and also provide the name for the project.



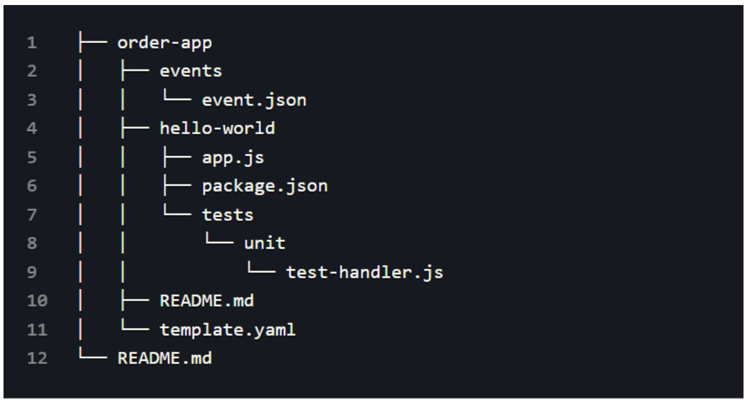
```
sudo yum install -y jq

export AWS_REGION=$(curl -s 169.254.169.254/latest/dynamic/instance-identity/document | jq -r '.region')
echo "export AWS_REGION=${AWS_REGION}" | tee -a ~/.bash_profile
echo "export ORDER_APP=order-app" | tee -a ~/.bash_profile
source ~/.bash_profile
```

SAM template for hello-world is cloned and then it will be modified to server purpose for our app. We will be creating a folder called order-app using the following code.



```
cd ~/environment

sam init \
    --name $ORDER_APP \
    --runtime nodejs14.x  \
    --dependency-manager npm \
    --app-template hello-world
```

The directory structure created is provided below:



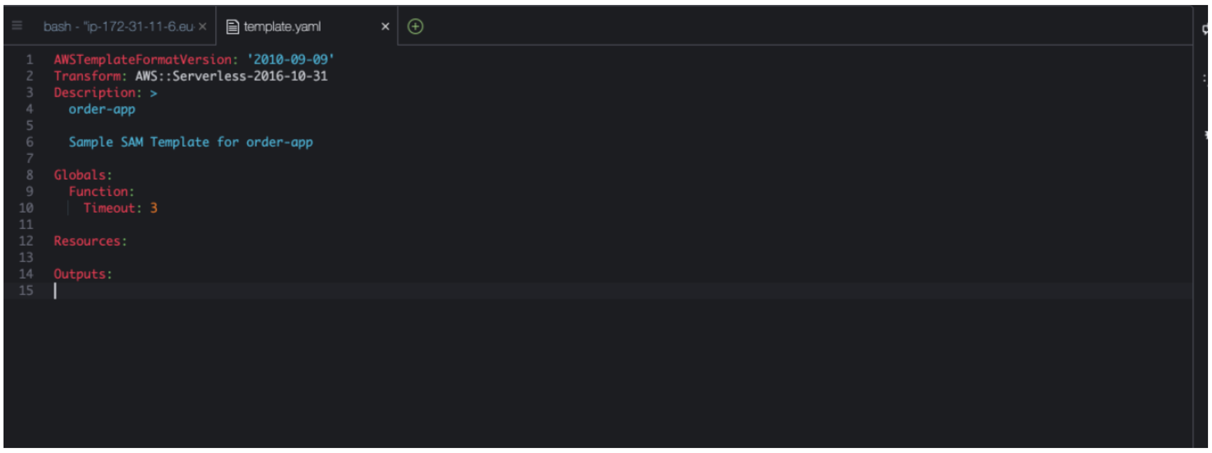
```
├── order-app
│   ├── events
│   │   └── event.json
│   ├── hello-world
│   │   ├── app.js
│   │   ├── package.json
│   │   └── tests
│   │       └── unit
│   │           └── test-handler.js
│   ├── README.md
│   └── template.yaml
└── README.md
```

Two folders are created which are:

Hello-world - This contains Lambda source code and tests written in Python. Returns a HTTP 200 status with Hello world message.

template.yaml every components that is to be deployed into AWS environment is stored here.

The template file looks as follows:



```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: >
  order-app

  Sample SAM Template for order-app

Globals:
  Function:
    Timeout: 3

Resources:

Outputs:

```

Build and Deploy the SAM project using following commands:

```
cd ~/environment/$ORDER_APP
sam build
```

```
sam deploy --stack-name $ORDER_APP --region $AWS_REGION --guided
```

4.1 Database Initialization

For our Order Service, we'll be storing our orders in Amazon DynamoDB. To get started, we'll construct a DynamoDB Table with the default settings.

The Partition Key will be user id, and the Range Key will be id in the DynamoDB table.

Partition key - A partition key is a simple primary key made up of only one property. The sort key or range key is used to sort objects that belong to the same

partition. For example, the partition key for an Order table would be CustomerId, and the sort key would be OrderId.

In the Cloud9 editor, open template.yaml. The DynamoDB Table resource will be added to the Resources node, as shown below.

```yaml
OrderTable:
  Type: AWS::DynamoDB::Table
  Properties:
    TableName: order-table
    AttributeDefinitions:
      - AttributeName: user_id
        AttributeType: S
      - AttributeName: id
        AttributeType: S
    KeySchema:
      - AttributeName: user_id
        KeyType: HASH
      - AttributeName: id
        KeyType: RANGE
    ProvisionedThroughput:
      ReadCapacityUnits: 5
      WriteCapacityUnits: 5
```

Then, in the Outputs area, add the following DynamoDB Table Name as an Output:

```yaml
  DynamoDBTableName:
    Description: "DynamoDB Table Name"
    Value: !Ref OrderTable
```

Construct and deploy our SAM template in Cloud9 Terminal to generate the DynamoDB Table in AWS.

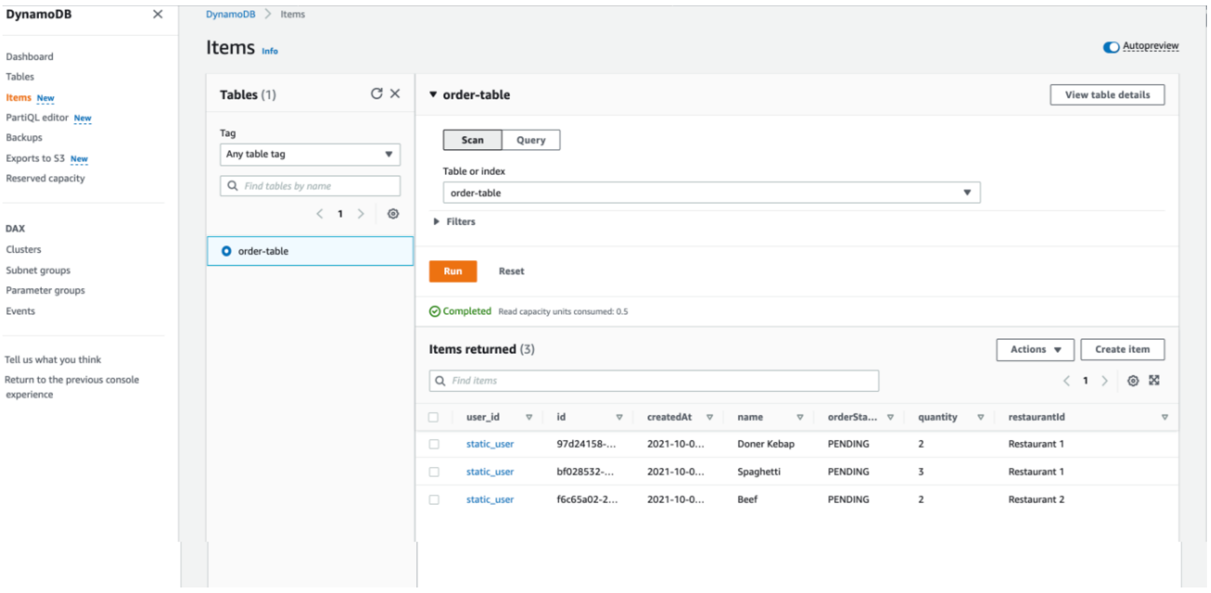
```
cd ~/environment/$ORDER_APP
sam build
sam deploy --no-confirm-changeset
```

```
aws dynamodb describe-table --table-name order-table --region $AWS_REGION
```

Let's add some sample data to our Order Table. Let's start by making a new folder and file for this (seed-orderdb.js). After that, install the requirements (node-uuid, and aws-sdk).

```
mkdir -p ~/environment/$ORDER_APP/populate-db
cd ~/environment/$ORDER_APP/populate-db
touch seed-orderdb.js

npm init -y
npm i aws-sdk node-uuid
```

After populating our database by using Python we can see the database below:



order-table

Completed Read capacity units consumed: 0.5

Items returned (3)

user_id	id	createdAt	name	orderSta...	quantity	restaurantId
static_user	97d24158-...	2021-10-0...	Doner Kebap	PENDING	2	Restaurant 1
static_user	bf028532-...	2021-10-0...	Spaghetti	PENDING	3	Restaurant 1
static_user	f6c65a02-2...	2021-10-0...	Beef	PENDING	2	Restaurant 2

The first component of our RESTful API will be /GET Orders, which will return a list of Orders. The get-orders lambda method will be written in Python.

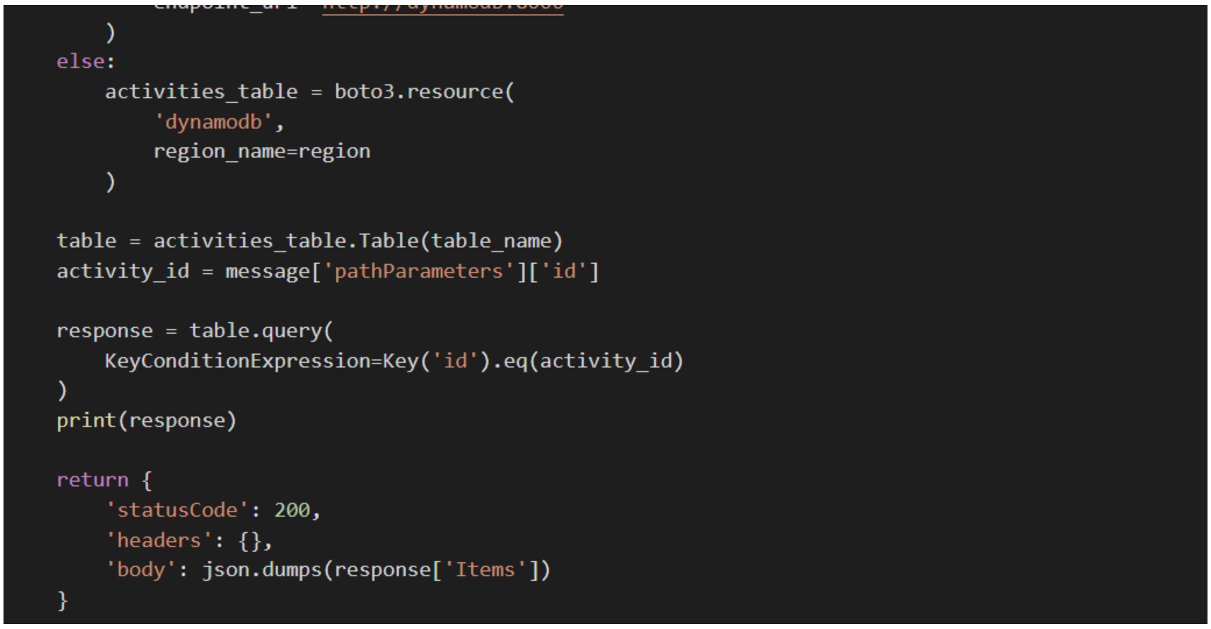
To develop a getOrders function, navigate to the order-app project and create the get-orders subdirectory.

```
cd ~/environment/$ORDER_APP
mkdir -p src/order-api/functions/get-orders
```

4.2 Creating API

In the Cloud9 editor, open the app.py file and the following code block is executed.

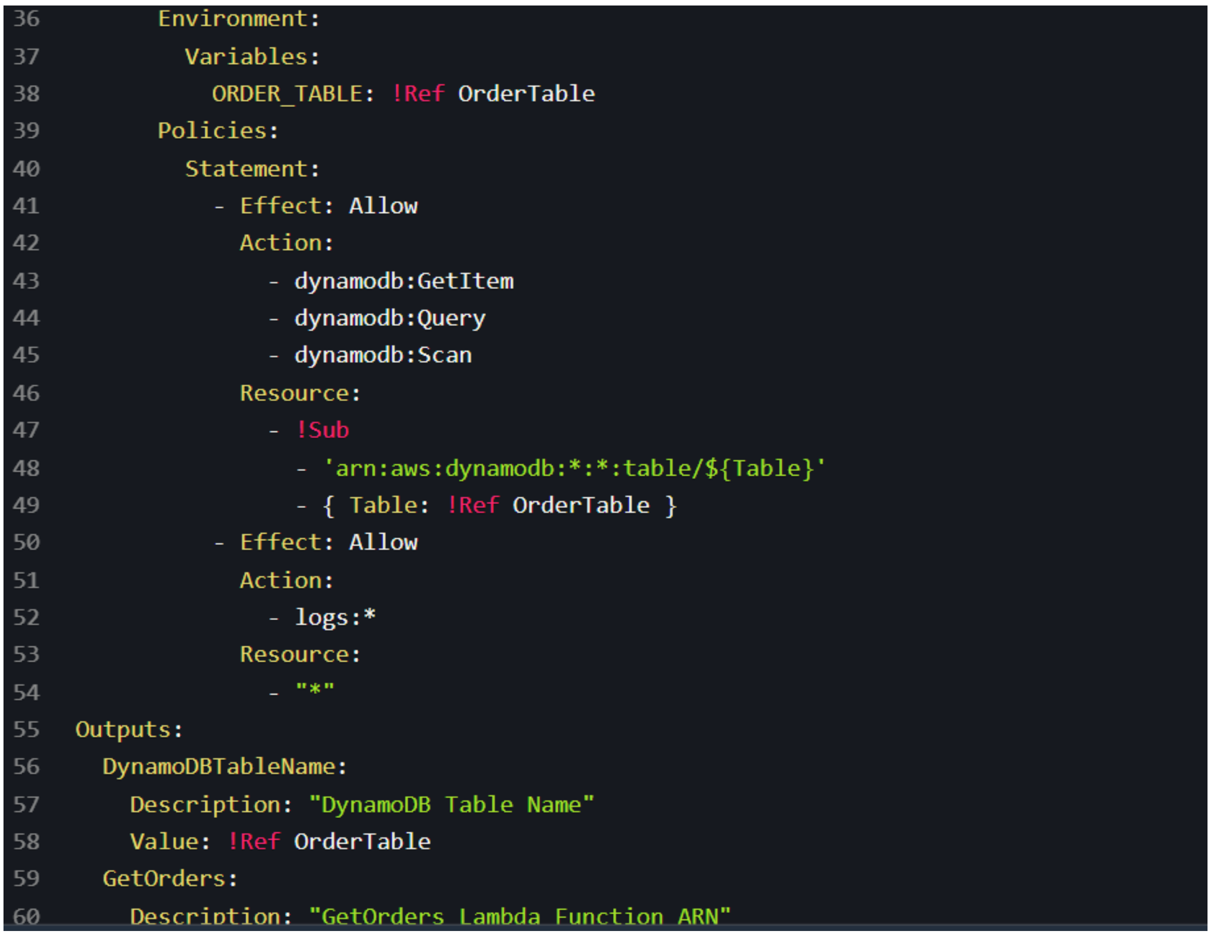
```
def lambda_handler(message, context):

    if ('pathParameters' not in message or
            message['httpMethod'] != 'GET'):
        return {
            'statusCode': 400,
            'headers': {},
            'body': json.dumps({'msg': 'Bad Request'})
        }

    table_name = os.environ.get('TABLE', 'Activities')
    region = os.environ.get('REGION', 'us-east-1')
    aws_environment = os.environ.get('AWSENV', 'AWS')

    if aws_environment == 'AWS_SAM_LOCAL':
        activities_table = boto3.resource(
            'dynamodb',
            endpoint_url='http://dynamodb:8000'
        )
    else:
        activities_table = boto3.resource(
            'dynamodb',
            region_name=region
        )

    table = activities_table.Table(table_name)
    activity_id = message['pathParameters']['id']
```



```python
        )
    else:
        activities_table = boto3.resource(
            'dynamodb',
            region_name=region
        )

    table = activities_table.Table(table_name)
    activity_id = message['pathParameters']['id']

    response = table.query(
        KeyConditionExpression=Key('id').eq(activity_id)
    )
    print(response)

    return {
        'statusCode': 200,
        'headers': {},
        'body': json.dumps(response['Items'])
    }
```

It's time to update the SAM template for the app.py Lambda function deployment. We'll create a Function resource and its output in the template.yaml file.

Execute the following resource into the template's Resources section.



```yaml
      Environment:
        Variables:
          ORDER_TABLE: !Ref OrderTable
      Policies:
        Statement:
          - Effect: Allow
            Action:
              - dynamodb:GetItem
              - dynamodb:Query
              - dynamodb:Scan
            Resource:
              - !Sub
                - 'arn:aws:dynamodb:*:*:table/${Table}'
                - { Table: !Ref OrderTable }
          - Effect: Allow
            Action:
              - logs:*
            Resource:
              - "*"
Outputs:
  DynamoDBTableName:
    Description: "DynamoDB Table Name"
    Value: !Ref OrderTable
  GetOrders:
    Description: "GetOrders Lambda Function ARN"
```

Creating an empty file for API Gateway methods.

```
cd ~/environment/$ORDER_APP
touch api.yaml
```

Execute the following OpenAPI definition for ApiGateway to GET resource in our api.yaml file.

```
---
swagger: "2.0"
info:
  version: "1.0"
  title: "order-app"
basePath: "/Dev"
schemes:
- "https"
paths:
  /orders:
    get:
      responses: {}
      x-amazon-apigateway-integration:
        credentials:
          Fn::GetAtt: [ ApiGwExecutionRole, Arn ]
        type: "aws_proxy"
        httpMethod: "POST"
        uri:
          Fn::Sub: arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-31/functions/${GetOrders.Arn}/invocations
```

The API definition above creates a GET method under the /orders resource, as well as a proxy to the GetOrders Lambda function.

As seen above, we'll be passing two resource outputs from the SAM template file name. ApiGwExecutionRole and GetOrders are the two. Arn

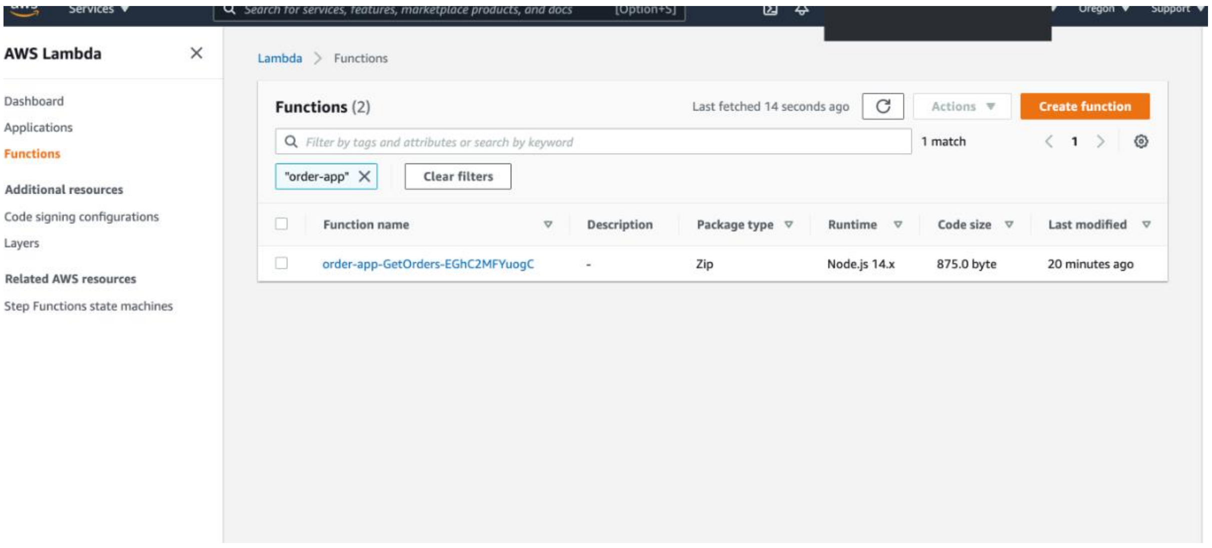
In the SAM template.yaml file, create API Gateway and APIGwExecutionRole resources.

Create and launch the SAM project: Create and launch the SAM project:

```
cd ~/environment/$ORDER_APP
sam build
sam deploy --no-confirm-changeset
```

The GetOrders function and API Gateway have now been deployed.

In the Lambda console, look at the newly formed getOrdersfunction:



Services ▼ Search for services, features, marketplace products, and docs [Option+S] [redacted] Oregon ▼ Support ▼

AWS Lambda

Dashboard
Applications
Functions

Additional resources
Code signing configurations
Layers

Related AWS resources
Step Functions state machines

Lambda > Functions

Functions (2) Last fetched 14 seconds ago Actions ▼ Create function

Filter by tags and attributes or search by keyword 1 match

"order-app" ✕ Clear filters

Function name	Description	Package type	Runtime	Code size	Last modified
order-app-GetOrders-EGhC2MFYuogC	-	Zip	Node.js 14.x	875.0 byte	20 minutes ago

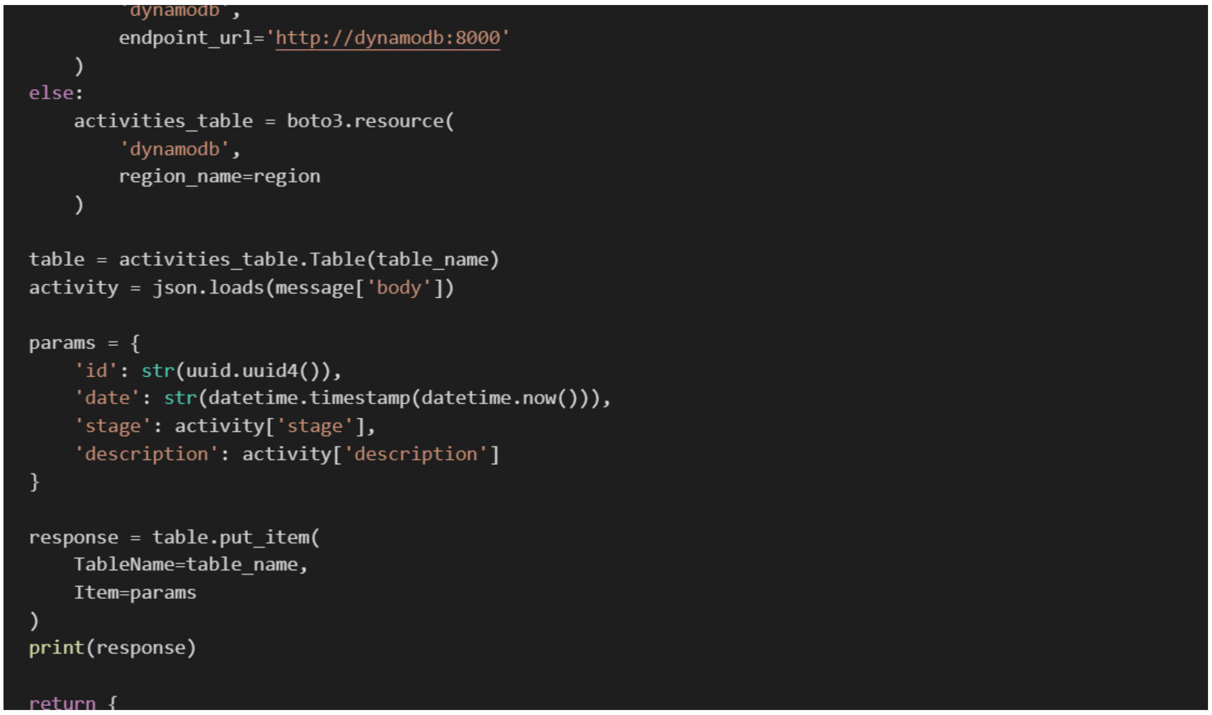
- Creating function

To develop a createOrder function, navigate to the order-app project and create the get-orders subdirectory.

```
cd ~/environment/$ORDER_APP
mkdir -p src/order-api/functions/post-orders
```

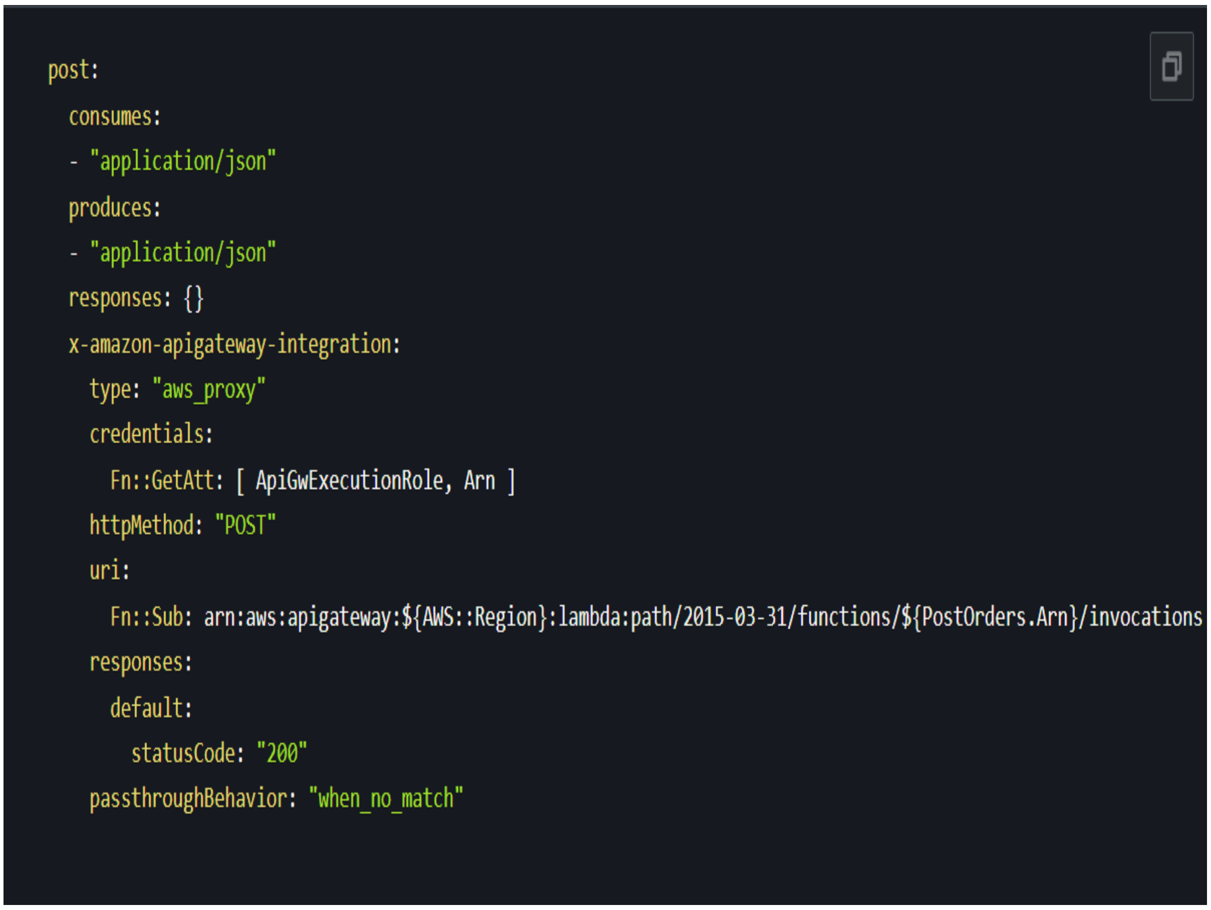
In the Cloud9 editor, open the app.py file and the following code block is executed.

```
def lambda_handler(message, context):

    if ('body' not in message or
            message['httpMethod'] != 'POST'):
        return {
            'statusCode': 400,
            'headers': {},
            'body': json.dumps({'msg': 'Bad Request'})
        }

    table_name = os.environ.get('TABLE', 'Activities')
    region = os.environ.get('REGION', 'us-east-1')
    aws_environment = os.environ.get('AWSENV', 'AWS')

    if aws_environment == 'AWS_SAM_LOCAL':
        activities_table = boto3.resource(
            'dynamodb',
            endpoint_url='http://dynamodb:8000'
        )
    else:
        activities_table = boto3.resource(
            'dynamodb',
            region_name=region
        )

    table = activities_table.Table(table_name)
```



```
        'dynamodb',
        endpoint_url='http://dynamodb:8000'
    )
else:
    activities_table = boto3.resource(
        'dynamodb',
        region_name=region
    )

table = activities_table.Table(table_name)
activity = json.loads(message['body'])

params = {
    'id': str(uuid.uuid4()),
    'date': str(datetime.timestamp(datetime.now())),
    'stage': activity['stage'],
    'description': activity['description']
}

response = table.put_item(
    TableName=table_name,
    Item=params
)
print(response)

return {
```

Execute the following OpenAPI definition for ApiGateway to GET resource in our api.yaml file.



```
post:
  consumes:
  - "application/json"
  produces:
  - "application/json"
  responses: {}
  x-amazon-apigateway-integration:
    type: "aws_proxy"
    credentials:
      Fn::GetAtt: [ ApiGwExecutionRole, Arn ]
    httpMethod: "POST"
    uri:
      Fn::Sub: arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-31/functions/${PostOrders.Arn}/invocations
    responses:
      default:
        statusCode: "200"
    passthroughBehavior: "when_no_match"
```

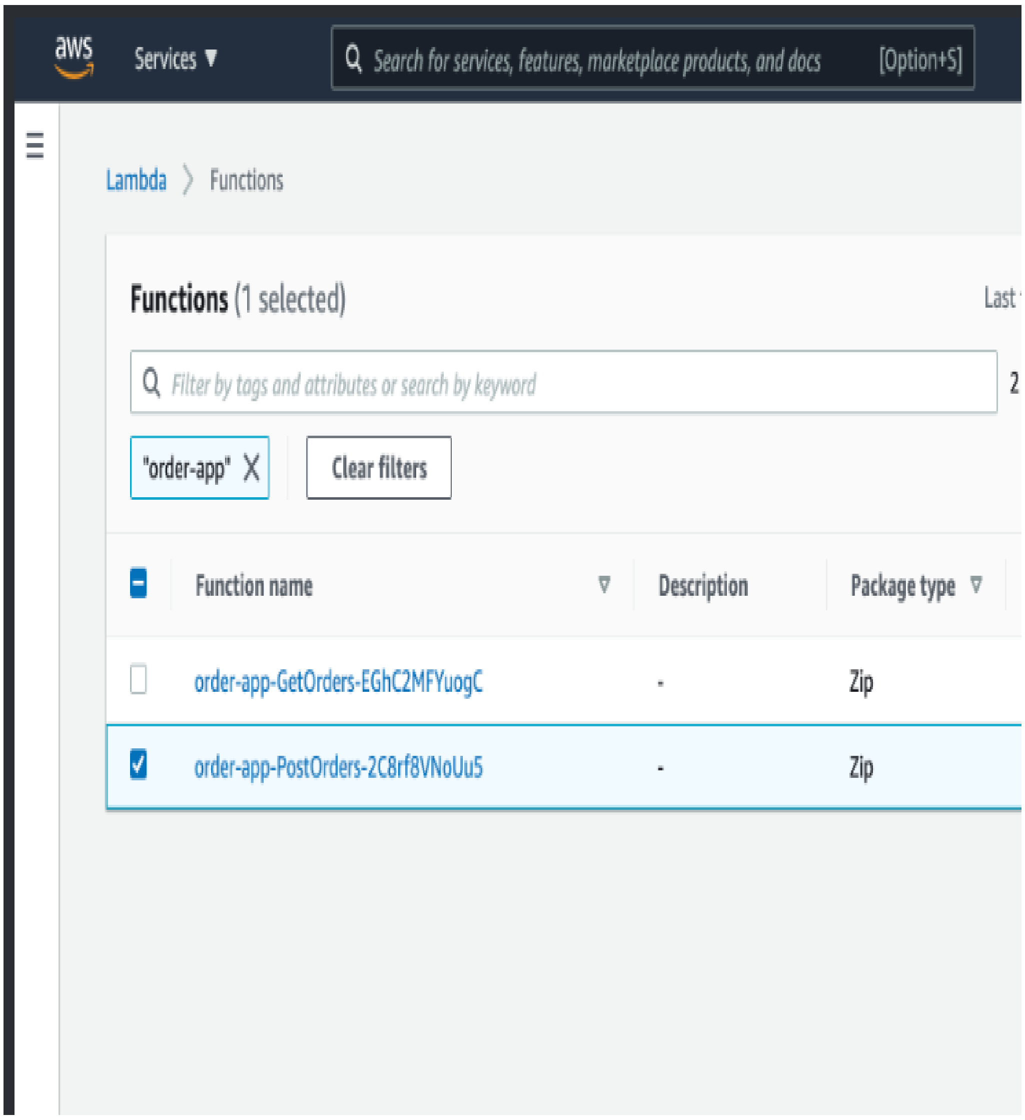
Finally, let's update ApiGwExecutionPolicy for PostOrders function to give an access in template.yaml file. Copy below condition into PolicyDocument Resource.

```
          - !GetAtt PostOrders.Arn
```

Build and Deploy the SAM project:

```
cd ~/environment/$ORDER_APP
sam build
sam deploy --no-confirm-changeset
```

The Post Orders Lambda function has now been deployed and linked with the API Gateway POST Method.
In the Lambda console, look at the newly formed postOrdersfunction:



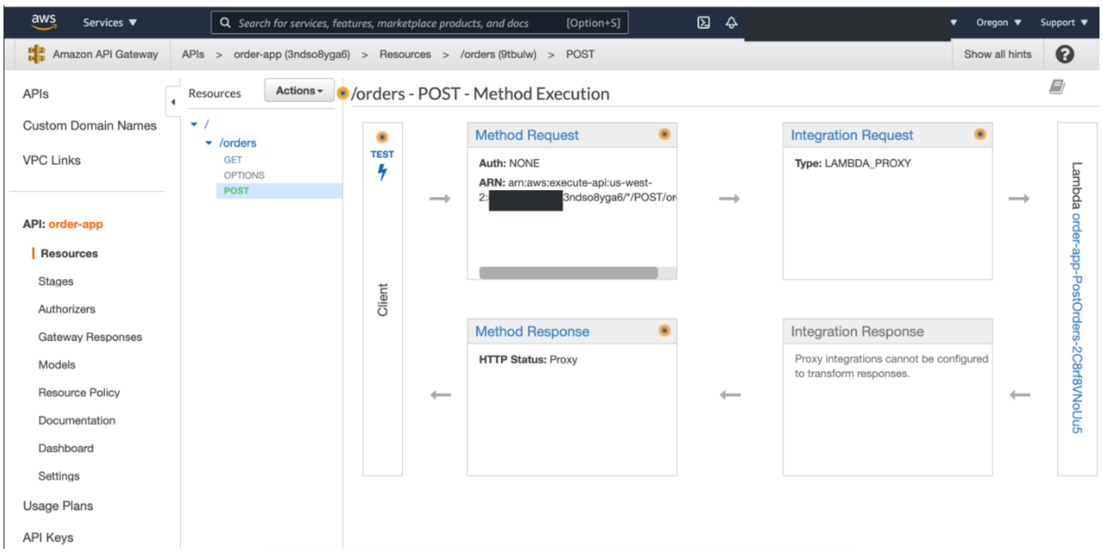
aws
Services ▼
Search for services, features, marketplace products, and docs [Option+S]
Lambda > Functions

Functions (1 selected)

Filter by tags and attributes or search by keyword

"order-app" X   Clear filters

	Function name	Description	Package type
☐	order-app-GetOrders-EGhC2MFYuogC	-	Zip
☑	order-app-PostOrders-2C8rf8VNoUu5	-	Zip



/orders - POST - Method Execution

Method Request
Auth: NONE
ARN: arn:aws:execute-api:us-west-2:[redacted]3ndso8yga6/*/POST/or

Integration Request
Type: LAMBDA_PROXY

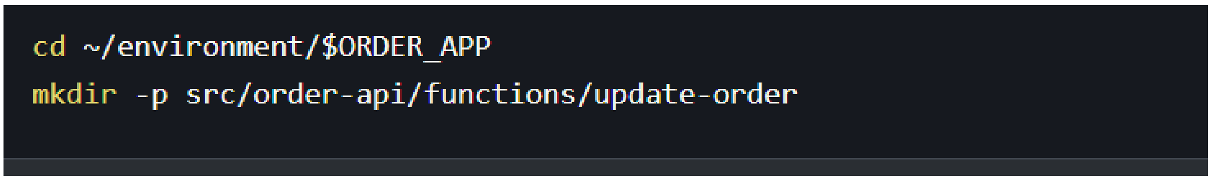
Method Response
HTTP Status: Proxy

Integration Response
Proxy integrations cannot be configured to transform responses.

Lambda order-app-PostOrders-2C8rf8VNoUu5

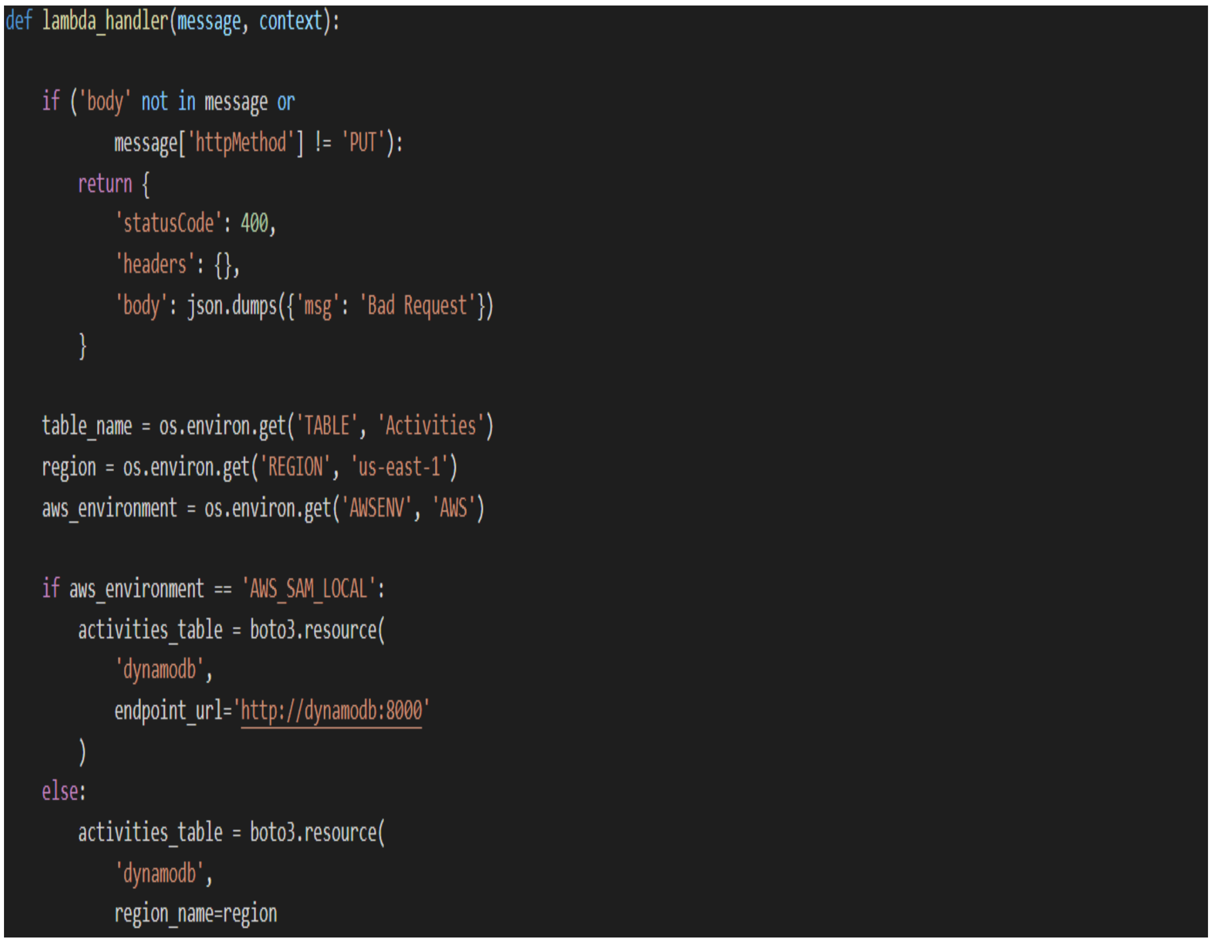
- Updating function

To develop a updateOrder function, navigate to the order-app project and create the get-orders subdirectory.



```
cd ~/environment/$ORDER_APP
mkdir -p src/order-api/functions/update-order
```

In the Cloud9 editor, open the app.py file and the following code block is executed.



```
def lambda_handler(message, context):

    if ('body' not in message or
            message['httpMethod'] != 'PUT'):
        return {
            'statusCode': 400,
            'headers': {},
            'body': json.dumps({'msg': 'Bad Request'})
        }

    table_name = os.environ.get('TABLE', 'Activities')
    region = os.environ.get('REGION', 'us-east-1')
    aws_environment = os.environ.get('AWSENV', 'AWS')

    if aws_environment == 'AWS_SAM_LOCAL':
        activities_table = boto3.resource(
            'dynamodb',
            endpoint_url='http://dynamodb:8000'
        )
    else:
        activities_table = boto3.resource(
            'dynamodb',
            region_name=region
```

Execute the following OpenAPI definition for ApiGateway to GET resource in our api.yaml file.

```yaml
Environment:
  Variables:
    ORDER_TABLE: !Ref OrderTable
Policies:
  Statement:
    - Effect: Allow
      Action:
        - dynamodb:UpdateItem
      Resource:
        - !Sub
          - 'arn:aws:dynamodb:*:*:table/${Table}'
          - { Table: !Ref OrderTable }
    - Effect: Allow
      Action:
        - logs:*
      Resource:
        - "*"
```

Finally, let's update ApiGwExecutionPolicy for updateOrders function to give an access in template.yaml file. Copy below condition into PolicyDocument Resource.

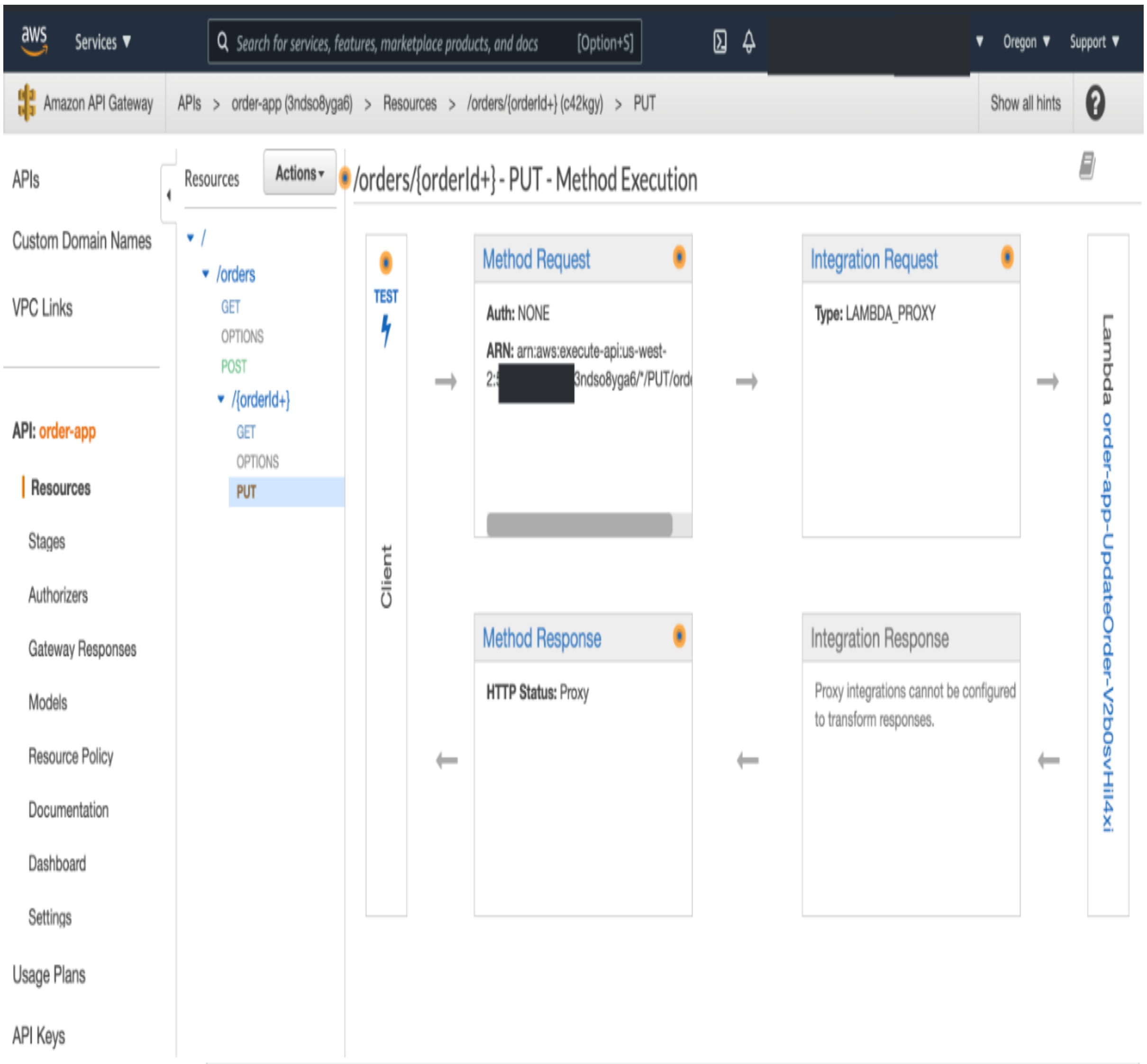
```yaml
put:
  produces:
  - "application/json"
  parameters:
  - name: "orderId"
    in: "path"
    required: true
    type: "string"
  responses: {}
  x-amazon-apigateway-integration:
    credentials:
      Fn::GetAtt: [ ApiGwExecutionRole, Arn ]
    httpMethod: "POST"
    uri:
      Fn::Sub: arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-31/functions/${UpdateOrder.Arn}/invocatio
    responses:
      default:
        statusCode: "200"
    passthroughBehavior: "when_no_match"
    type: "aws_proxy"
```

```
            - !GetAtt UpdateOrder.Arn
```

Build and Deploy the SAM project:

```
cd ~/environment/$ORDER_APP
sam build
sam deploy --no-confirm-changeset
```

The Update Orders Lambda function has now been deployed and linked with the API Gateway POST Method.
In the Lambda console, look at the newly formed updateOrdersfunction:



/orders/{orderId+} - PUT - Method Execution

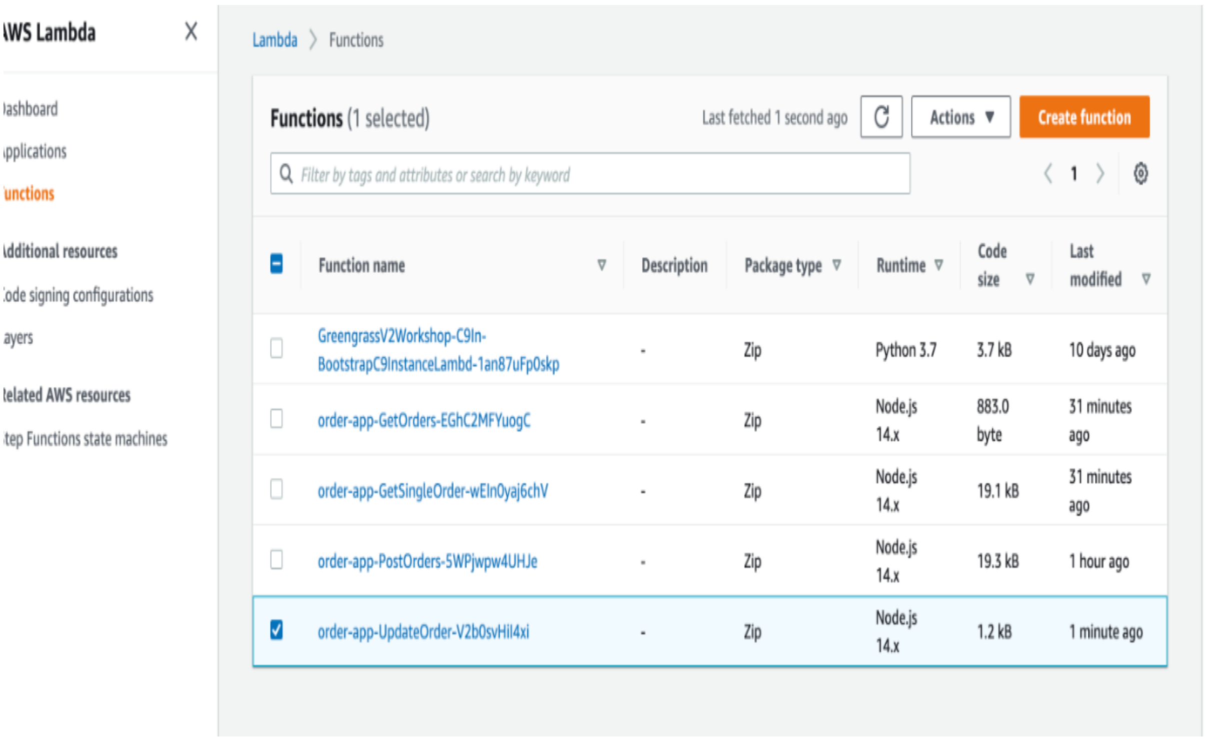
Auth: NONE

ARN: arn:aws:execute-api:us-west-2:[redacted]3ndso8yga6/*/PUT/ord

Type: LAMBDA_PROXY

HTTP Status: Proxy

Proxy integrations cannot be configured to transform responses.


WS Lambda
ashboard
pplications
unctions
dditional resources
ode signing configurations
ayers
elated AWS resources
tep Functions state machines
Lambda > Functions
Last fetched 1 second ago
Actions
Create function
Filter by tags and attributes or search by keyword

Functions (1 selected)

	Function name	Description	Package type	Runtime	Code size	Last modified
☐	GreengrassV2Workshop-C9In-BootstrapC9InstanceLambd-1an87uFp0skp	-	Zip	Python 3.7	3.7 kB	10 days ago
☐	order-app-GetOrders-EGhC2MFYuogC	-	Zip	Node.js 14.x	883.0 byte	31 minutes ago
☐	order-app-GetSingleOrder-wEIn0yaj6chV	-	Zip	Node.js 14.x	19.1 kB	31 minutes ago
☐	order-app-PostOrders-5WPjwpw4UHJe	-	Zip	Node.js 14.x	19.3 kB	1 hour ago
☑	order-app-UpdateOrder-V2b0svHil4xi	-	Zip	Node.js 14.x	1.2 kB	1 minute ago

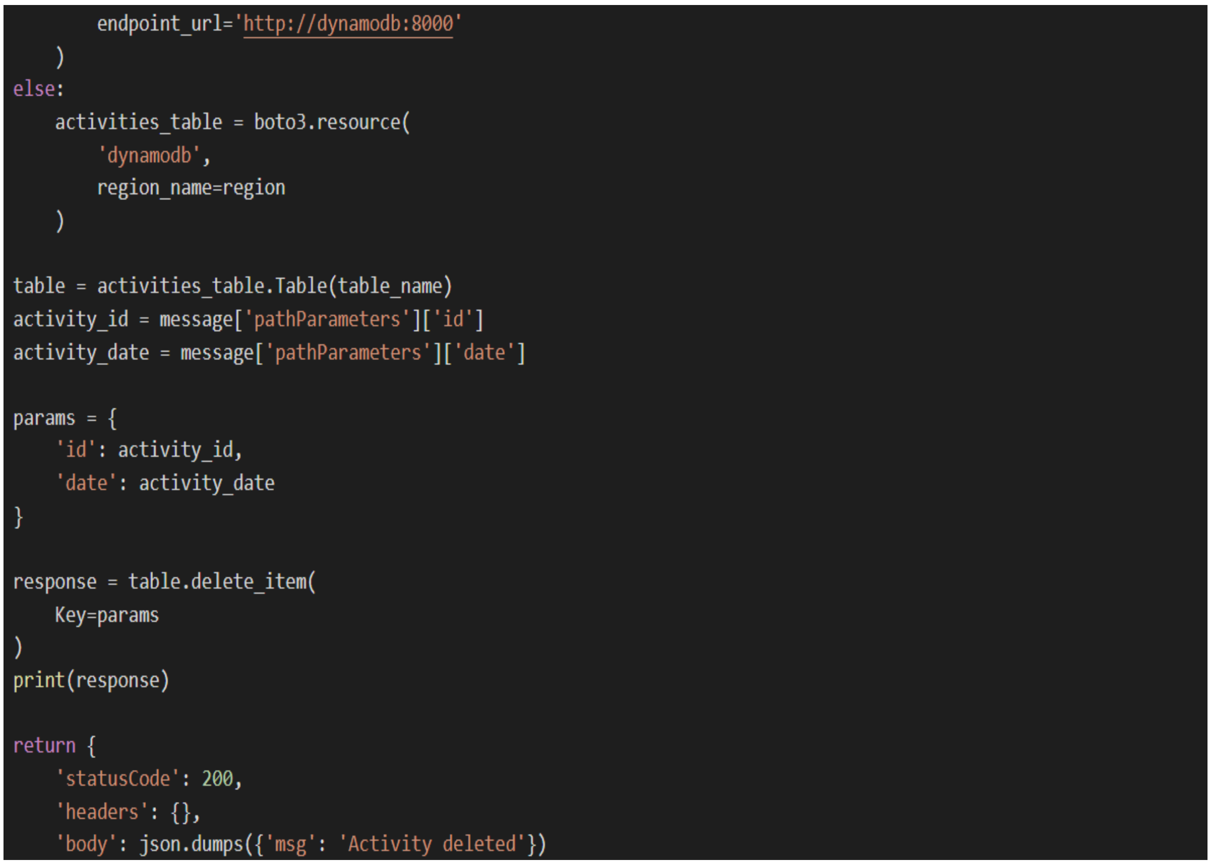
- Deleting function

To develop a deleteOrder function, navigate to the order-app project and create the get-orders subdirectory.

```
cd ~/environment/$ORDER_APP
mkdir -p src/order-api/functions/delete-order
```

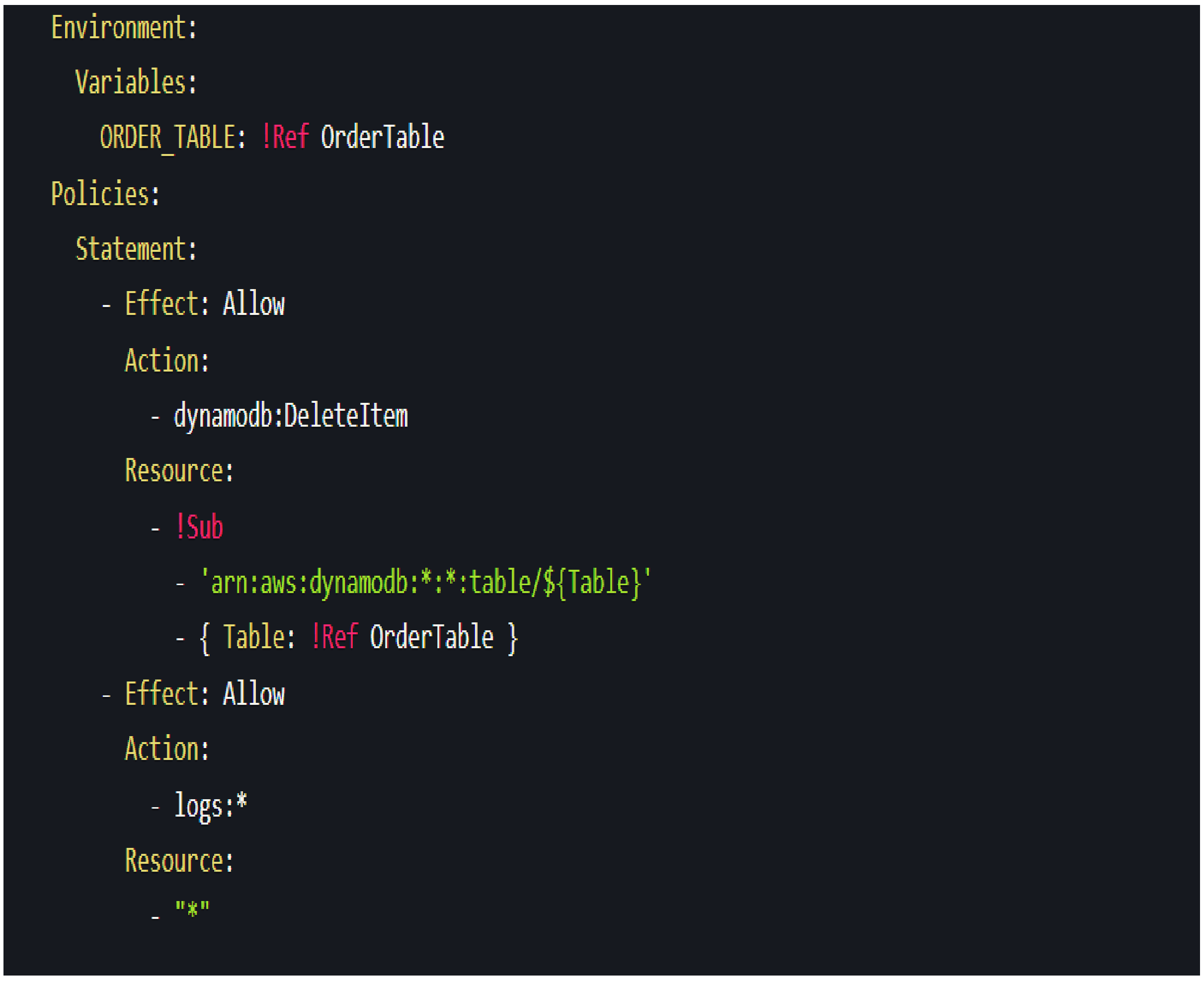
In the Cloud9 editor, open the app.py file and the following code block is executed.

```
def lambda_handler(message, context):

    if ('pathParameters' not in message or
            message['httpMethod'] != 'DELETE'):
        return {
            'statusCode': 400,
            'headers': {},
            'body': json.dumps({'msg': 'Bad Request'})
        }

    table_name = os.environ.get('TABLE', 'Activities')
    region = os.environ.get('REGION', 'us-east-1')
    aws_environment = os.environ.get('AWSENV', 'AWS')

    if aws_environment == 'AWS_SAM_LOCAL':
        activities_table = boto3.resource(
            'dynamodb',
            endpoint_url='http://dynamodb:8000'
        )
```

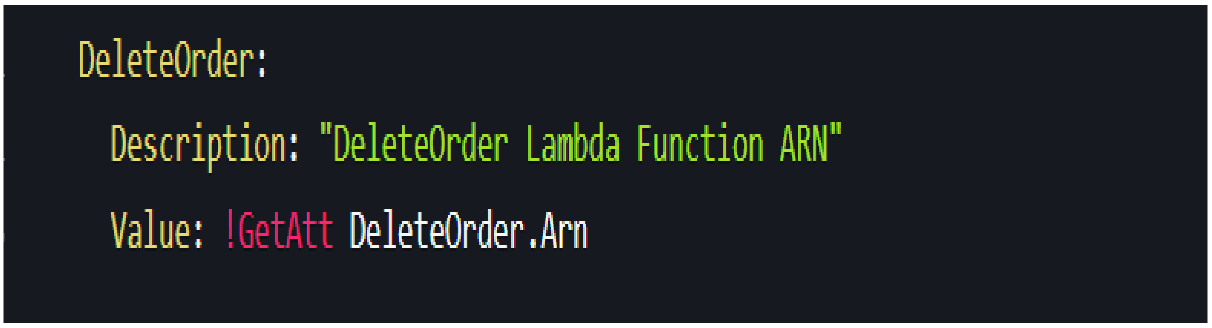


```python
        endpoint_url='http://dynamodb:8000'
    )
else:
    activities_table = boto3.resource(
        'dynamodb',
        region_name=region
    )

table = activities_table.Table(table_name)
activity_id = message['pathParameters']['id']
activity_date = message['pathParameters']['date']

params = {
    'id': activity_id,
    'date': activity_date
}

response = table.delete_item(
    Key=params
)
print(response)

return {
    'statusCode': 200,
    'headers': {},
    'body': json.dumps({'msg': 'Activity deleted'})
```

Execute the following OpenAPI definition for ApiGateway to GET resource in our api.yaml file.

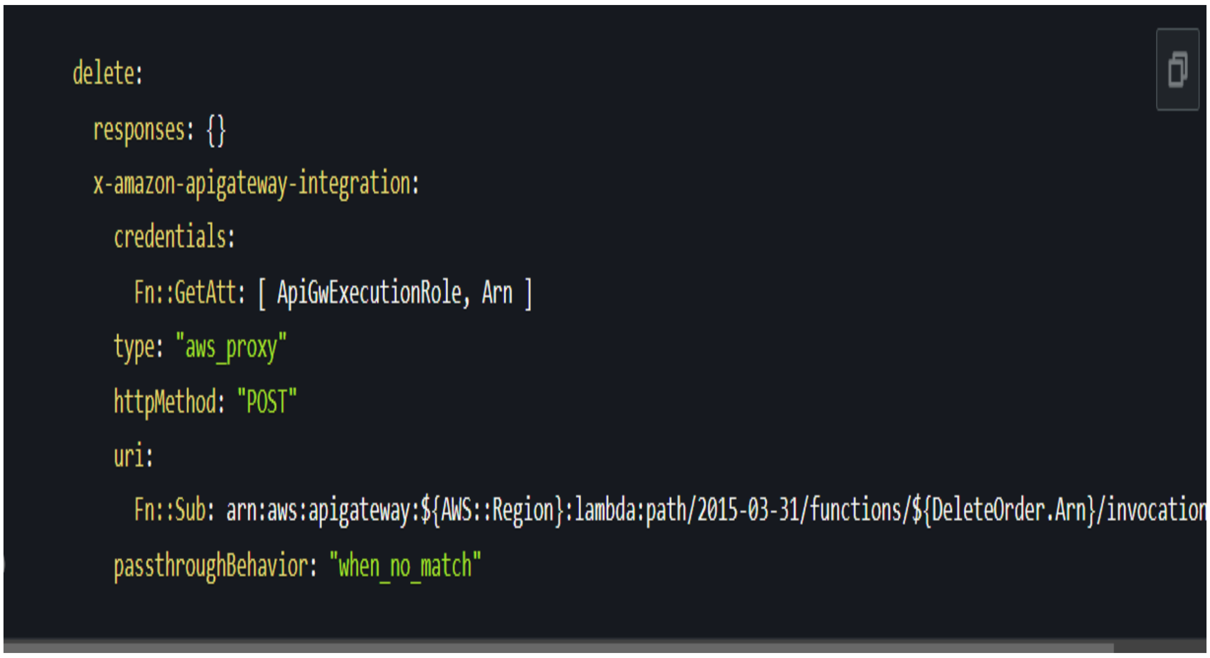


```yaml
Environment:
  Variables:
    ORDER_TABLE: !Ref OrderTable
Policies:
  Statement:
    - Effect: Allow
      Action:
        - dynamodb:DeleteItem
      Resource:
        - !Sub
          - 'arn:aws:dynamodb:*:*:table/${Table}'
          - { Table: !Ref OrderTable }
    - Effect: Allow
      Action:
        - logs:*
      Resource:
        - "*"
```

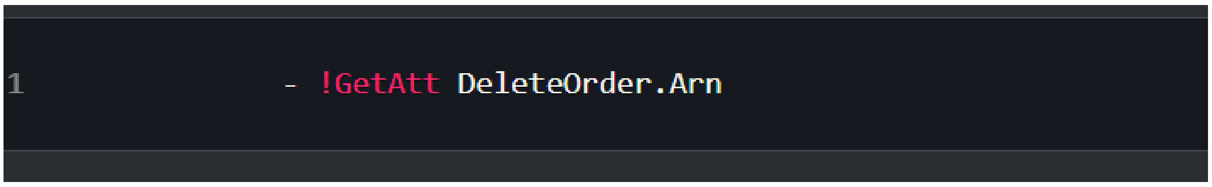


```yaml
DeleteOrder:
  Description: "DeleteOrder Lambda Function ARN"
  Value: !GetAtt DeleteOrder.Arn
```

Finally, let's update ApiGwExecutionPolicy for deleteOrders function to give an access in template.yaml file. Copy below condition into PolicyDocument Resource.

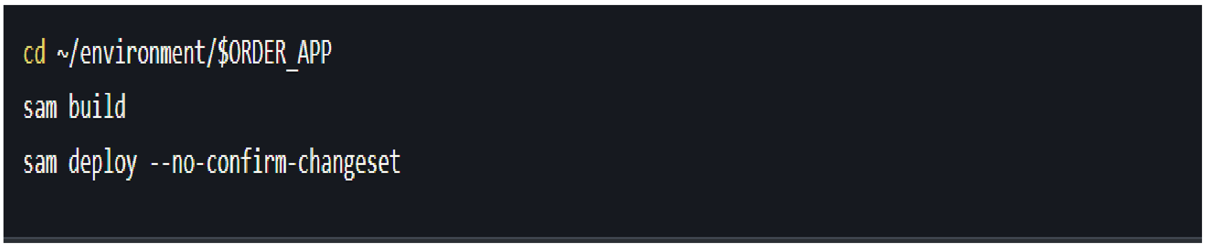


```
delete:
  responses: {}
  x-amazon-apigateway-integration:
    credentials:
      Fn::GetAtt: [ ApiGwExecutionRole, Arn ]
    type: "aws_proxy"
    httpMethod: "POST"
    uri:
      Fn::Sub: arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-31/functions/${DeleteOrder.Arn}/invocatio
    passthroughBehavior: "when_no_match"
```



```
              - !GetAtt DeleteOrder.Arn
```

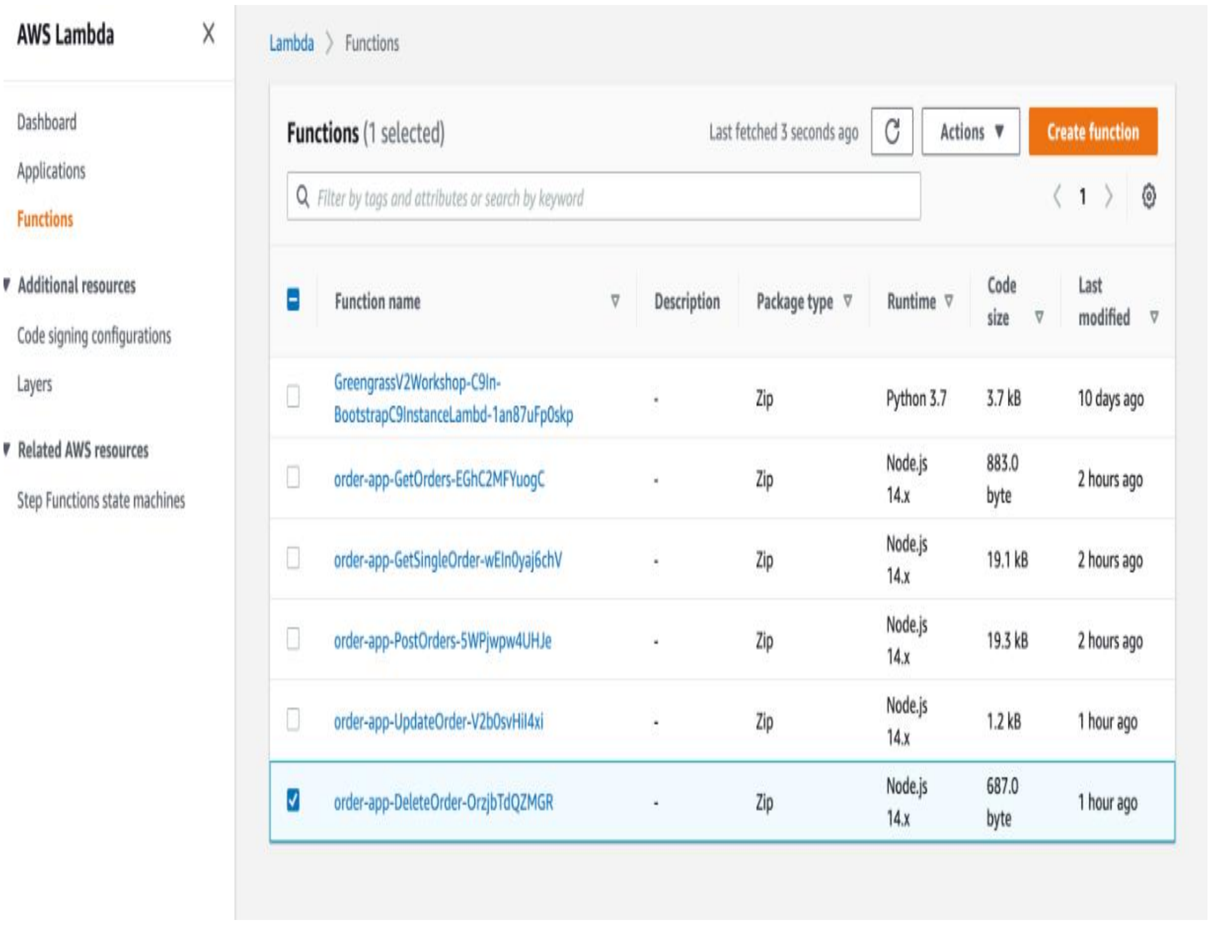
Build and Deploy the SAM project:



```
cd ~/environment/$ORDER_APP
sam build
sam deploy --no-confirm-changeset
```

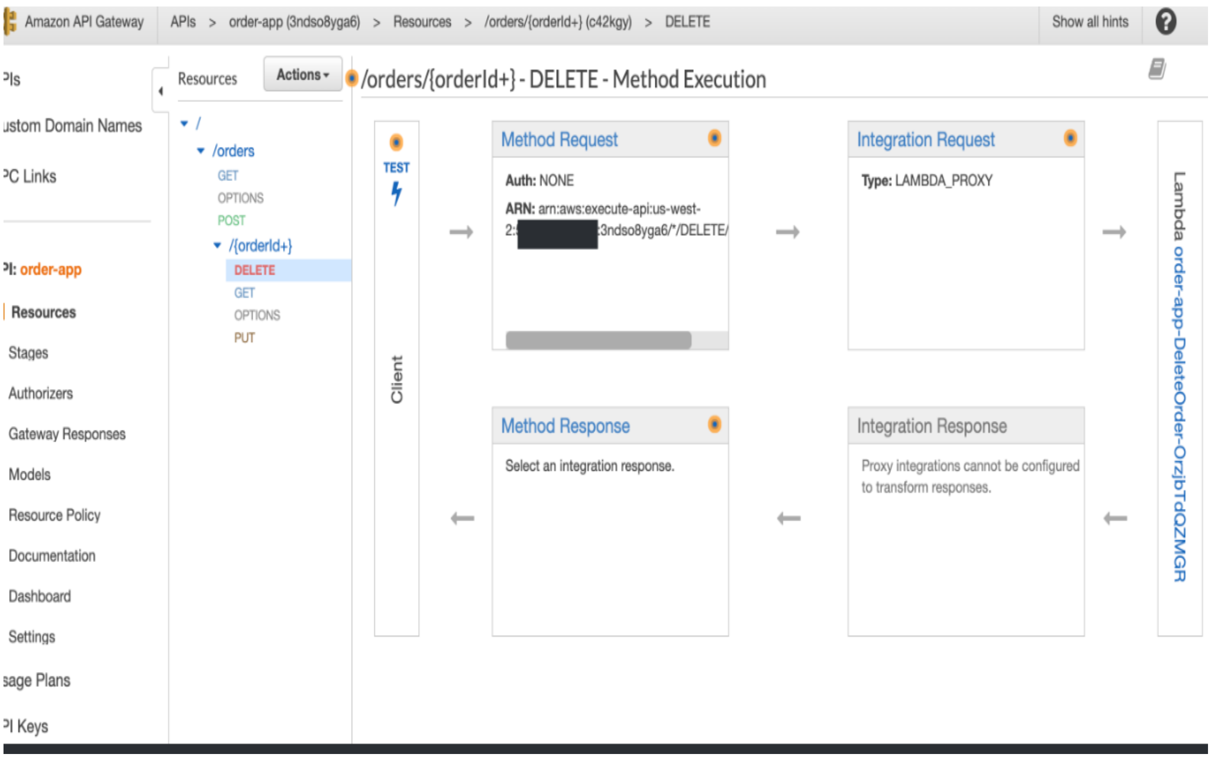
The Delete Orders Lambda function has now been deployed and linked with the API Gateway POST Method.

In the Lambda console, look at the newly formed deleteOrdersfunction:


AWS Lambda
Dashboard
Applications
Functions
Additional resources
Code signing configurations
Layers
Related AWS resources
Step Functions state machines
Lambda > Functions
Last fetched 3 seconds ago
Actions
Create function
Filter by tags and attributes or search by keyword

Functions (1 selected)

	Function name	Description	Package type	Runtime	Code size	Last modified
☐	GreengrassV2Workshop-C9In-BootstrapC9InstanceLambd-1an87uFp0skp	-	Zip	Python 3.7	3.7 kB	10 days ago
☐	order-app-GetOrders-EGhC2MFYuogC	-	Zip	Node.js 14.x	883.0 byte	2 hours ago
☐	order-app-GetSingleOrder-wEln0yaj6chV	-	Zip	Node.js 14.x	19.1 kB	2 hours ago
☐	order-app-PostOrders-5WPjwpw4UHJe	-	Zip	Node.js 14.x	19.3 kB	2 hours ago
☐	order-app-UpdateOrder-V2b0svHil4xi	-	Zip	Node.js 14.x	1.2 kB	1 hour ago
✓	order-app-DeleteOrder-OrzjbTdQZMGR	-	Zip	Node.js 14.x	687.0 byte	1 hour ago



/orders/{orderId+} - DELETE - Method Execution

Method Request
Auth: NONE
ARN: arn:aws:execute-api:us-west-2:[redacted]:3ndso8yga6/*/DELETE/

Integration Request
Type: LAMBDA_PROXY

Method Response
Select an integration response.

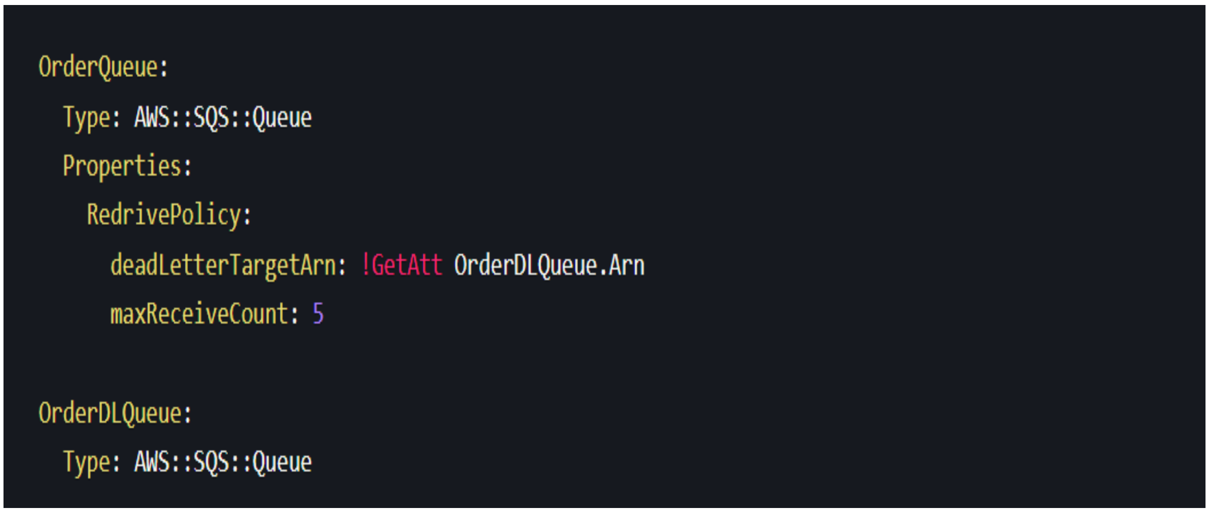
Integration Response
Proxy integrations cannot be configured to transform responses.

4.3 Order Poller Addition

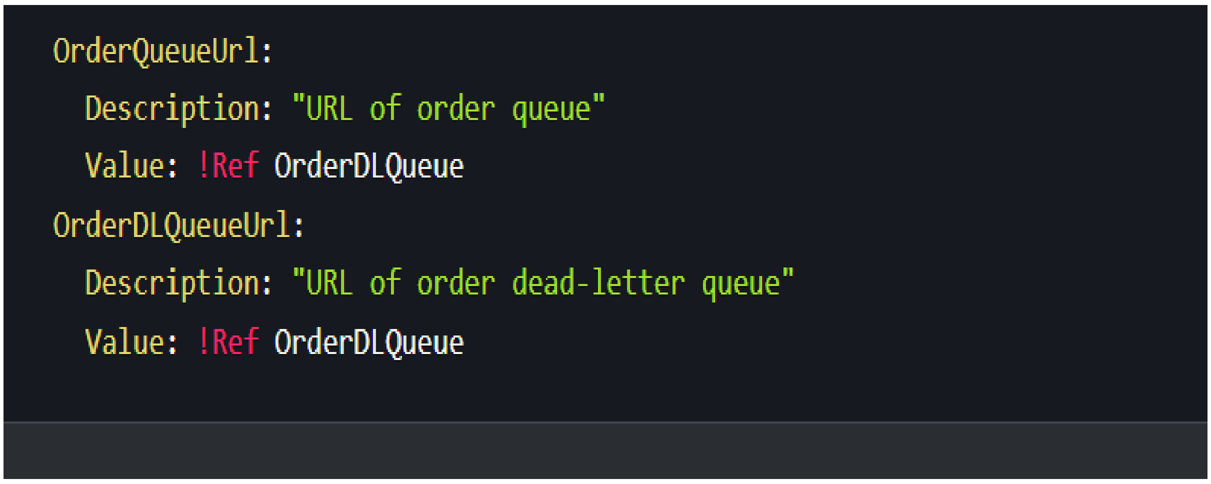
We'll create a SQS Queue with a Dead Letter Queue in this section, then add the Order Post function as an event source to the SQS Queue.

To construct SQS Queues, add Queue and Deadletter Queue to the template.yaml file. We'll set maxReceiveCount to 5 to see how many messages we can process before sending them to the Deadletter queue.

Copy the resources listed below into the Resources section of the template.yaml file.



```yaml
OrderQueue:
  Type: AWS::SQS::Queue
  Properties:
    RedrivePolicy:
      deadLetterTargetArn: !GetAtt OrderDLQueue.Arn
      maxReceiveCount: 5

OrderDLQueue:
  Type: AWS::SQS::Queue
```



```yaml
OrderQueueUrl:
  Description: "URL of order queue"
  Value: !Ref OrderDLQueue
OrderDLQueueUrl:
  Description: "URL of order dead-letter queue"
  Value: !Ref OrderDLQueue
```

To poll orders from SQS, add SQS as an event source to Post Orders Lambda. Now, in template.yaml's PostOrders resource, add the following Events block in the same line as Environment and Policies.

```yaml
Events:
  SQSEventSource:
    Type: SQS
    Properties:
      Queue: !GetAtt OrderQueue.Arn
      BatchSize: 10
```

Update the Post Orders function to read SQS order messages.

Instead of utilising the uuid library for the Order's Id, we'll use the messageId generated by SQS. Then, to read batches of messages from SQS, a for loop will be added.

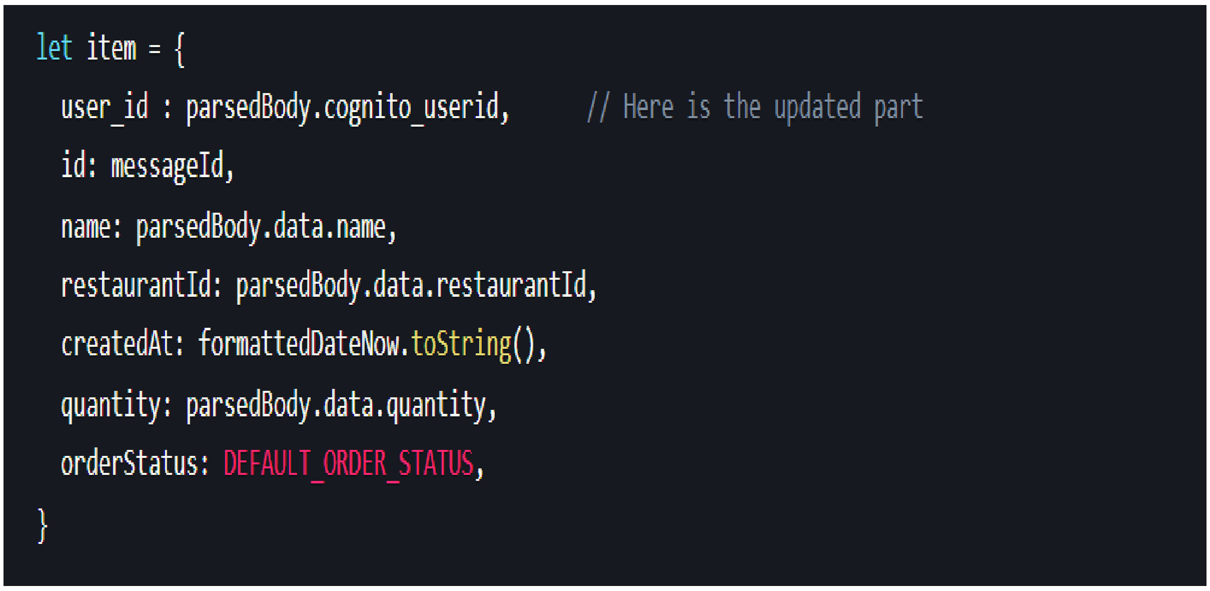
Go to the post-orders directory, which contains the app.js file.

```
cd ~/environment/$ORDER_APP/src/order-api/functions/post-orders/
```

4.4 Authentication Phase

Authentication

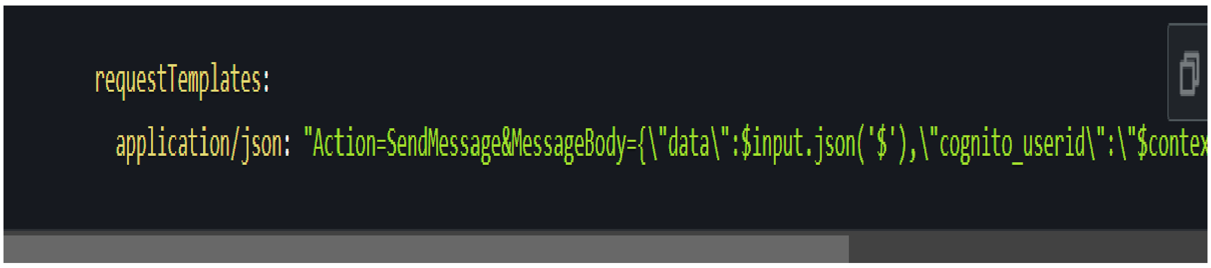
Open the app.js file in the Cloud9 editor at /environment/$ORDER APP/src/order-api/functions/post-orders, then replace the following item object with the previous one as shown below.



```
let item = {
  user_id : parsedBody.cognito_userid,      // Here is the updated part
  id: messageId,
  name: parsedBody.data.name,
  restaurantId: parsedBody.data.restaurantId,
  createdAt: formattedDateNow.toString(),
  quantity: parsedBody.data.quantity,
  orderStatus: DEFAULT_ORDER_STATUS,
}
```

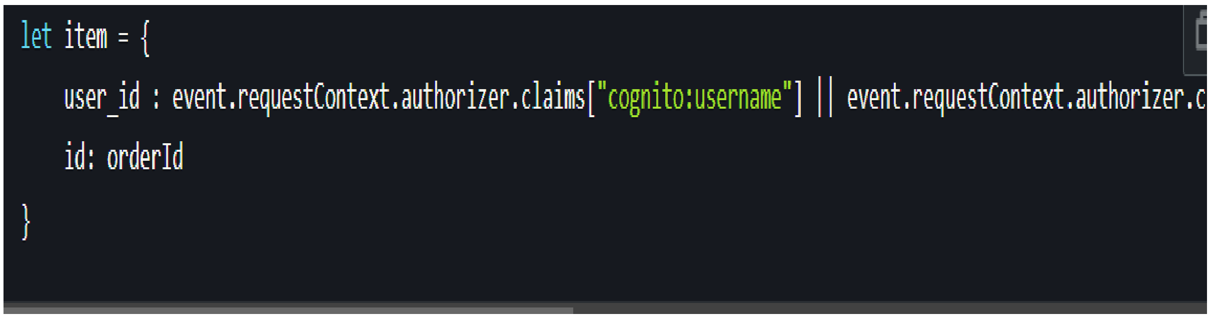
Cognito delivers the cognito authorization parameters to API Gateway in a requestTemplate when we integrate Cognito with API Gateway in lab1. The postOrders function in this request template will now include the Cognito userid and username.

Open the api.yaml file in the Cloud9 editor and add cognito parameters to MessageBody in requestTemplates in the POST section of the /environment/$ORDER APP/api.yaml file.



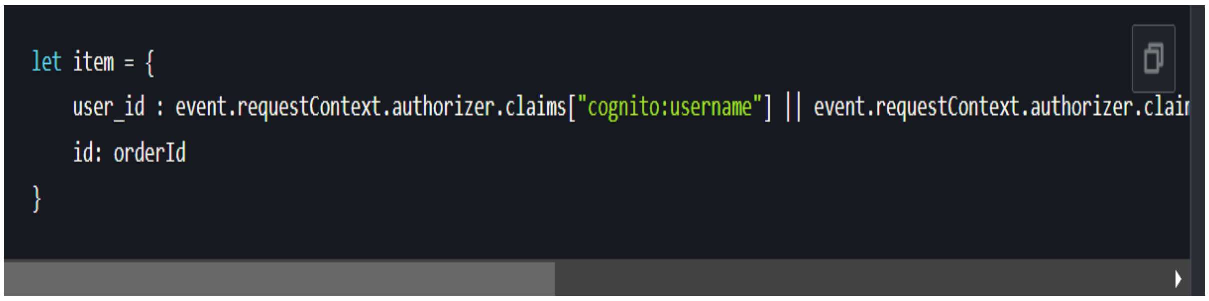
```
    requestTemplates:
      application/json: "Action=SendMessage&MessageBody={\"data\":$input.json('$'),\"cognito_userid\":\"$contex
```

Update the function 'Update Order': Open the app.js file in the Cloud9 editor at /environment/$ORDER APP/src/order-api/functions/update-order, then replace the following item object with the previous one as shown below.



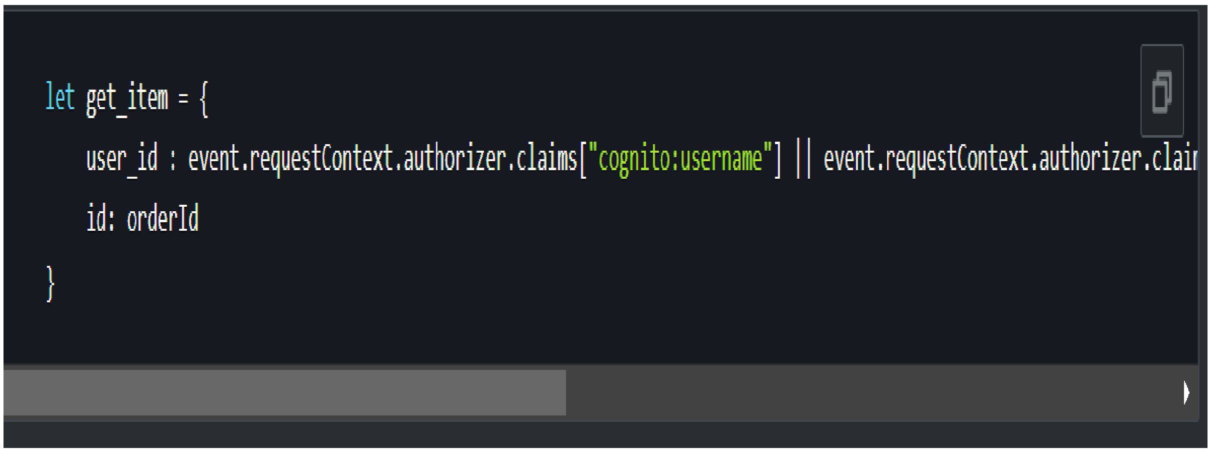
```
let item = {
    user_id : event.requestContext.authorizer.claims["cognito:username"] || event.requestContext.authorizer.c
    id: orderId
}
```

'Delete Order' has been updated: Open the app.js file in the Cloud9 editor at /environment/$ORDER APP/src/order-api/functions/delete-order, then replace the following item object with the previous one as shown below.



```
let item = {
    user_id : event.requestContext.authorizer.claims["cognito:username"] || event.requestContext.authorizer.clair
    id: orderId
}
```

'Get Single Order' has been updated: Open the app.js file in the Cloud9 editor at /environment/$ORDER APP/src/order-api/functions/get-single-order, then change the following item object with the previous one as shown below.



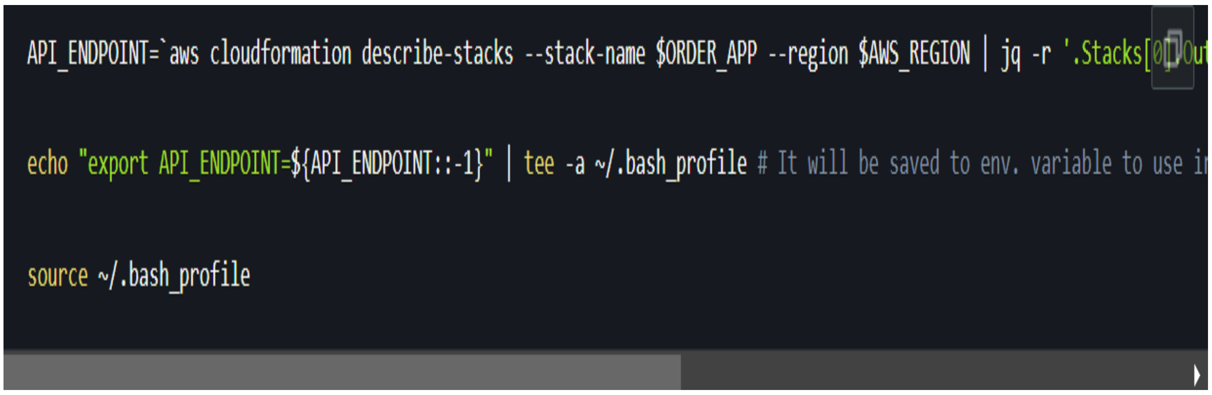
```
let get_item = {
    user_id : event.requestContext.authorizer.claims["cognito:username"] || event.requestContext.authorizer.clair
    id: orderId
}
```

CHAPTER 05: CONCLUSIONS

5.1 Testing Results

Triggering the API Endpoint after installing the API gateway and GetOrders Function in this section. The API endpoint can be found in the SAM output.

Extracting the API ENDPOINT from SAM and store it as an environment variable:

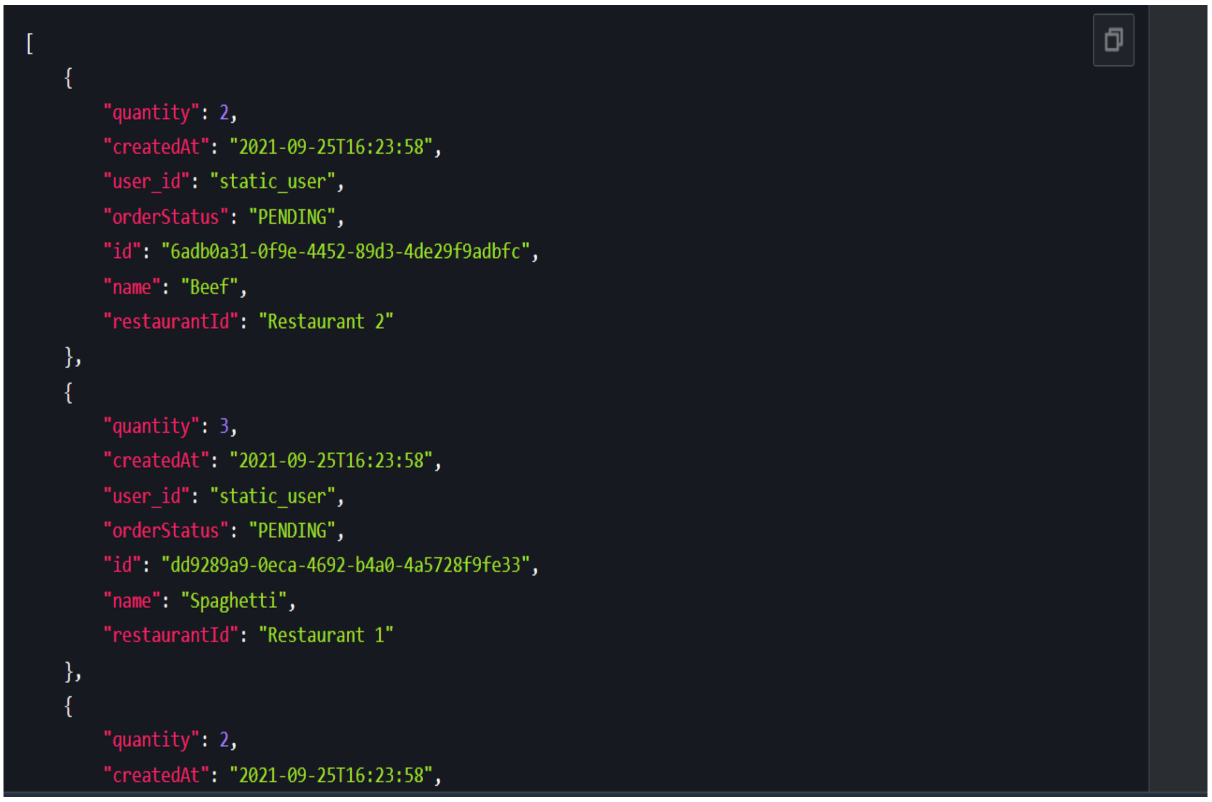


```
API_ENDPOINT=`aws cloudformation describe-stacks --stack-name $ORDER_APP --region $AWS_REGION | jq -r '.Stacks[0].Out

echo "export API_ENDPOINT=${API_ENDPOINT::-1}" | tee -a ~/.bash_profile # It will be saved to env. variable to use i

source ~/.bash_profile
```

Curl:

curl -s $API_ENDPOINT | python3 -m json.tool

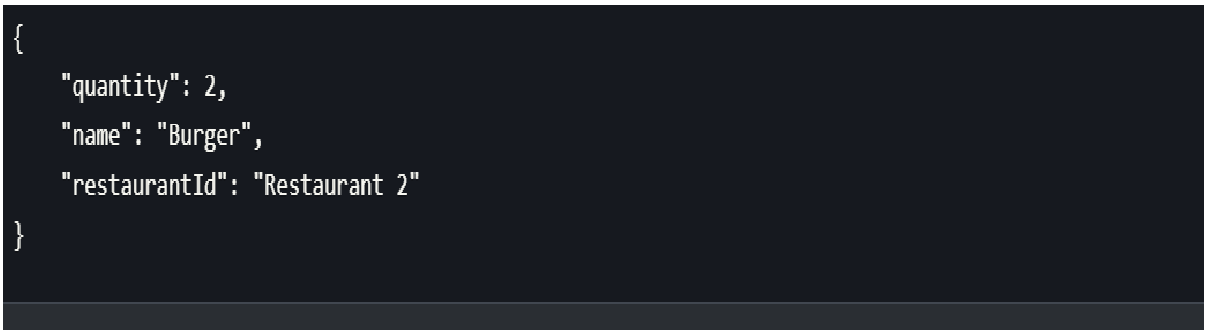
Output:



```
[
    {
        "quantity": 2,
        "createdAt": "2021-09-25T16:23:58",
        "user_id": "static_user",
        "orderStatus": "PENDING",
        "id": "6adb0a31-0f9e-4452-89d3-4de29f9adbfc",
        "name": "Beef",
        "restaurantId": "Restaurant 2"
    },
    {
        "quantity": 3,
        "createdAt": "2021-09-25T16:23:58",
        "user_id": "static_user",
        "orderStatus": "PENDING",
        "id": "dd9289a9-0eca-4692-b4a0-4a5728f9fe33",
        "name": "Spaghetti",
        "restaurantId": "Restaurant 1"
    },
    {
        "quantity": 2,
        "createdAt": "2021-09-25T16:23:58",
```

Triggering the API Endpoint after installing the API gateway and PostOrders Function in this section. The API endpoint can be found in the SAM output.

creating an Order now. Sending a JSON Payload to our API Gateway.
Sample Payload:

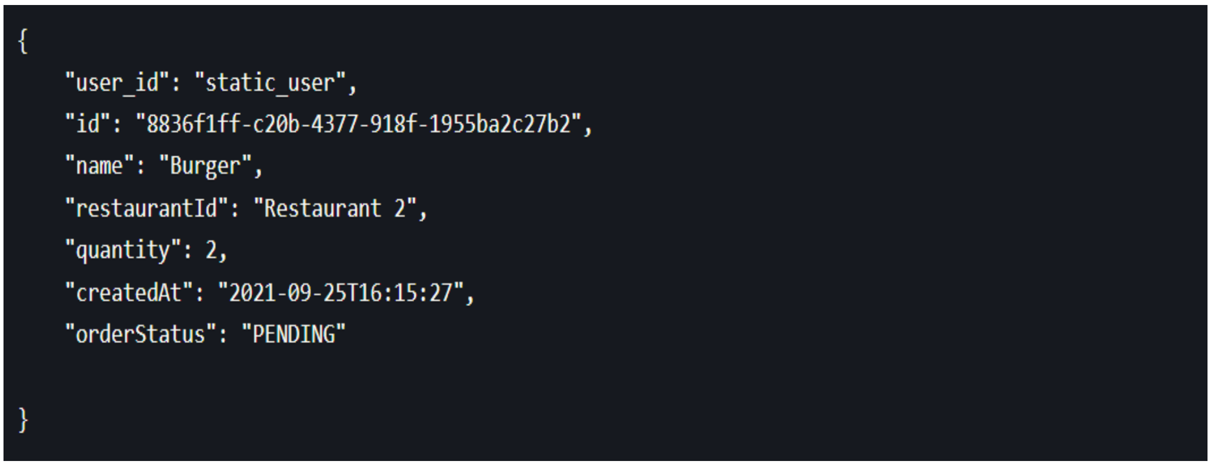


```
{
    "quantity": 2,
    "name": "Burger",
    "restaurantId": "Restaurant 2"
}
```

Curl:

```
curl -s --header "Content-Type: application/json" \
  --request POST \
  --data '{"name":"Burger","restaurantId":"Restaurant 2","quantity":2 }' \
  $API_ENDPOINT | python3 -m json.tool
```

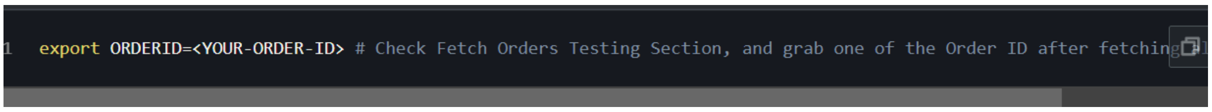
Output:



```
{
    "user_id": "static_user",
    "id": "8836f1ff-c20b-4377-918f-1955ba2c27b2",
    "name": "Burger",
    "restaurantId": "Restaurant 2",
    "quantity": 2,
    "createdAt": "2021-09-25T16:15:27",
    "orderStatus": "PENDING"

}
```

Triggering the API Endpoint after installing the API gateway and UpdateOrders Function in this section. The API endpoint can be found in the SAM output.

Replace the OrderID into <YOUR-ORDER-ID> field:

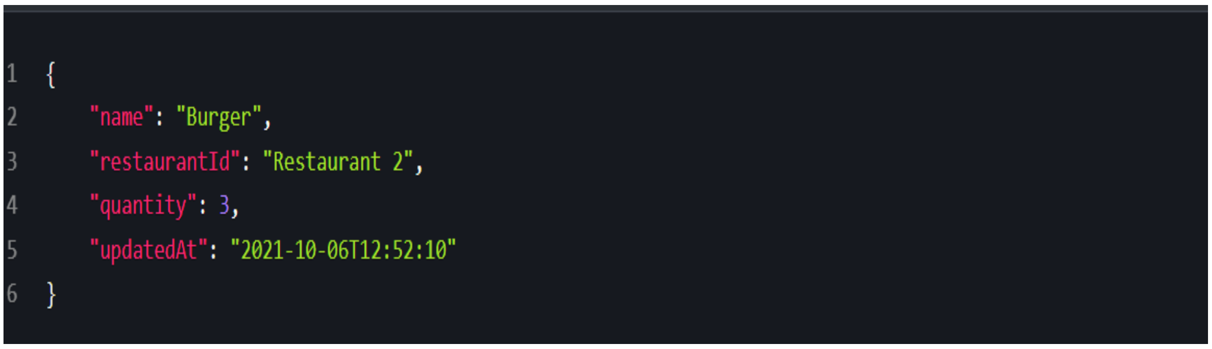


```
export ORDERID=<YOUR-ORDER-ID> # Check Fetch Orders Testing Section, and grab one of the Order ID after fetching
```

Curl:

```
curl -s --header "Content-Type: application/json" \
  --request PUT \
  --data '{"name":"Sushi","restaurantId":"Fancy Restaurant","quantity":12 }' \
  $API_ENDPOINT/$ORDERID | python3 -m json.tool
```

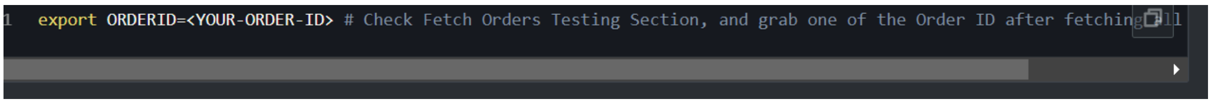
Output:



```
{
    "name": "Burger",
    "restaurantId": "Restaurant 2",
    "quantity": 3,
    "updatedAt": "2021-10-06T12:52:10"
}
```

Triggering the API Endpoint after installing the API gateway and deleteOrders Function in this section. The API endpoint can be found in the SAM output.

Replace the OrderID into <YOUR-ORDER-ID> field:



```
export ORDERID=<YOUR-ORDER-ID> # Check Fetch Orders Testing Section, and grab one of the Order ID after fetching
```

Curl:

curl -I -s --request DELETE $API_ENDPOINT/$ORDERID

Output:

```
HTTP/2 204
.
.
.
```

Creating another Order. API Gateway will submit the order to SQS as a message at this time. The Lambda consumer will then poll it, process it, and store the results in DynamoDB.

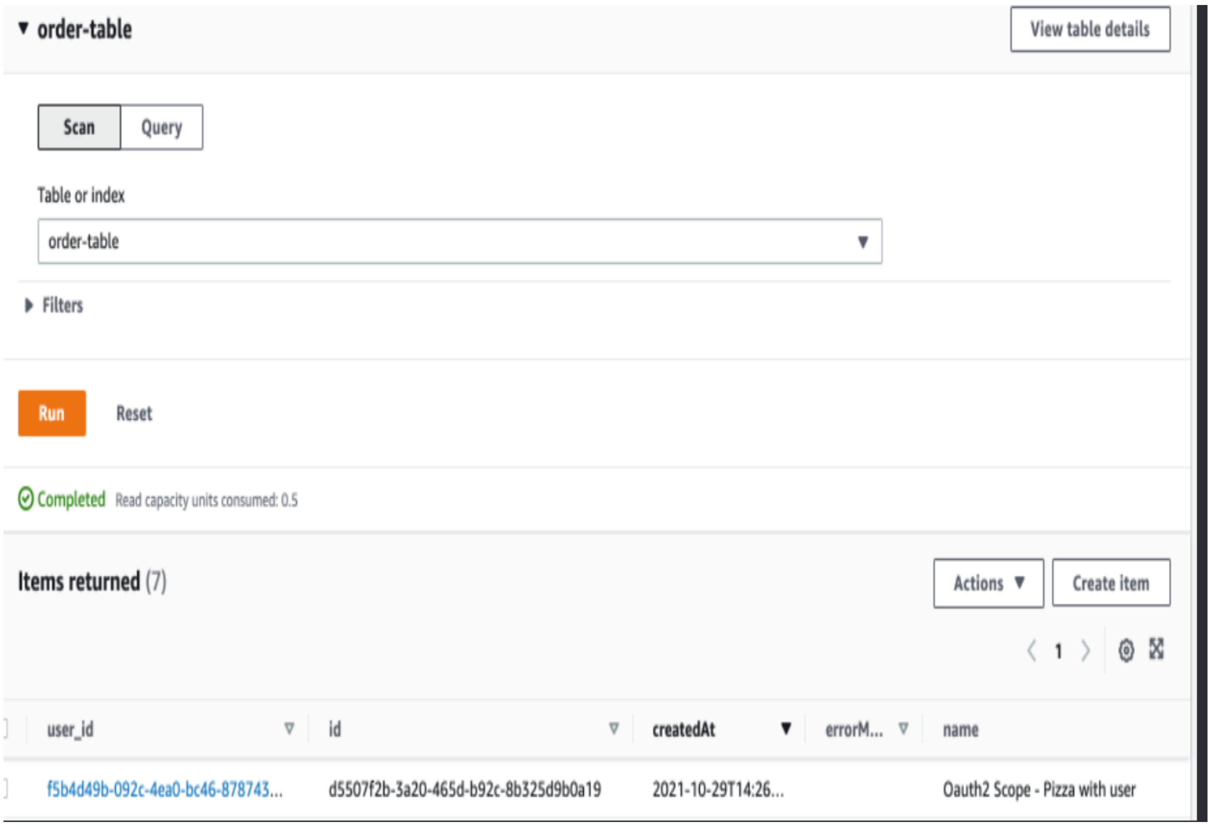
```
curl -s --header "Content-Type: application/json" \
  --request POST \
  --data '{"name":"Pizza","restaurantId":"Restaurant 99","quantity":3 }' \
  $API_ENDPOINT | python3 -m json.tool
```

Output:

```
{
    "SendMessageResponse": {
        "ResponseMetadata": {
            "RequestId": "4cef435b-a878-50e4-9d72-1a91c5a02d4a"
        },
        "SendMessageResult": {
            "MD5OfMessageAttributes": null,
            "MD5OfMessageBody": "c94d45bc611e259517988901a8b65ec5",
            "MD5OfMessageSystemAttributes": null,
            "MessageId": "53e2e439-d785-4332-9ed8-327bd0640cac",
            "SequenceNumber": null
        }
    }
}
```

Let's make a new Order with OAuth2 scope to remove.

```
curl -s --header "Content-Type: application/json" \
  --header "Authorization: Bearer $IDTOKEN" \
  --data '{ "name" : "Oauth2 Scope - Pizza with user", "restaurantId" : "Restaurant 11199", "quantity":3 }' \
  --request POST $API_ENDPOINT | python3 -m json.tool
```



user_id	id	createdAt	errorM...	name
f5b4d49b-092c-4ea0-bc46-878743...	d5507f2b-3a20-465d-b92c-8b325d9b0a19	2021-10-29T14:26...		Oauth2 Scope - Pizza with user

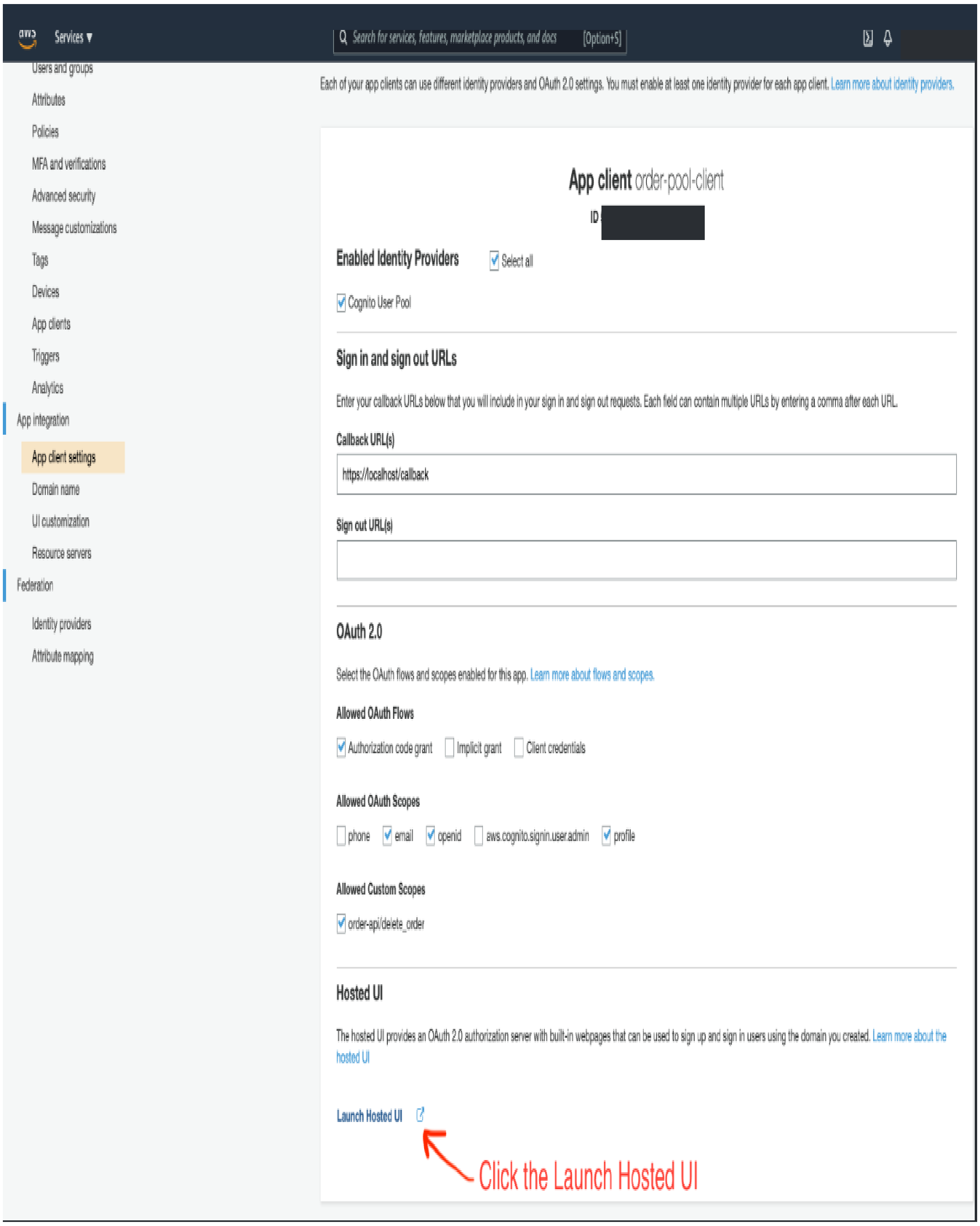
```
export ORDERID=<YOUR-ORDER-ID>
```

```
curl -I -s --header "Content-Type: application/json" \
  --header "Authorization: Bearer $IDTOKEN" \
  --request DELETE $API_ENDPOINT/$ORDERID
```

Respond with an HTTP 401 is provided Unauthorized error. Because the delete order scope is set to the Delete method.

We must visit the Cognito domain using delete scope in order to obtain an Access token with Oauth2 scope. The Cognito Domain name can be found in the Cognito console or SAM Output. However, because the response code argument is required, we must login to HostedUI in Lab4 with our Username and Password, then copy the response code and paste it at the end of the callbackurl.

In Cognito Pool's App Client Settings, click the Hosted UI button:



Each of your app clients can use different identity providers and OAuth 2.0 settings. You must enable at least one identity provider for each app client. Learn more about identity providers.

App client order-pool-client

ID [redacted]

Enabled Identity Providers [✓ Select all]

[✓ Cognito User Pool]

Sign in and sign out URLs

Enter your callback URLs below that you will include in your sign in and sign out requests. Each field can contain multiple URLs by entering a comma after each URL.

Callback URL(s)

https://localhost/callback

Sign out URL(s)

OAuth 2.0

Select the OAuth flows and scopes enabled for this app. Learn more about flows and scopes.

Allowed OAuth Flows

[✓ Authorization code grant] Implicit grant Client credentials

Allowed OAuth Scopes

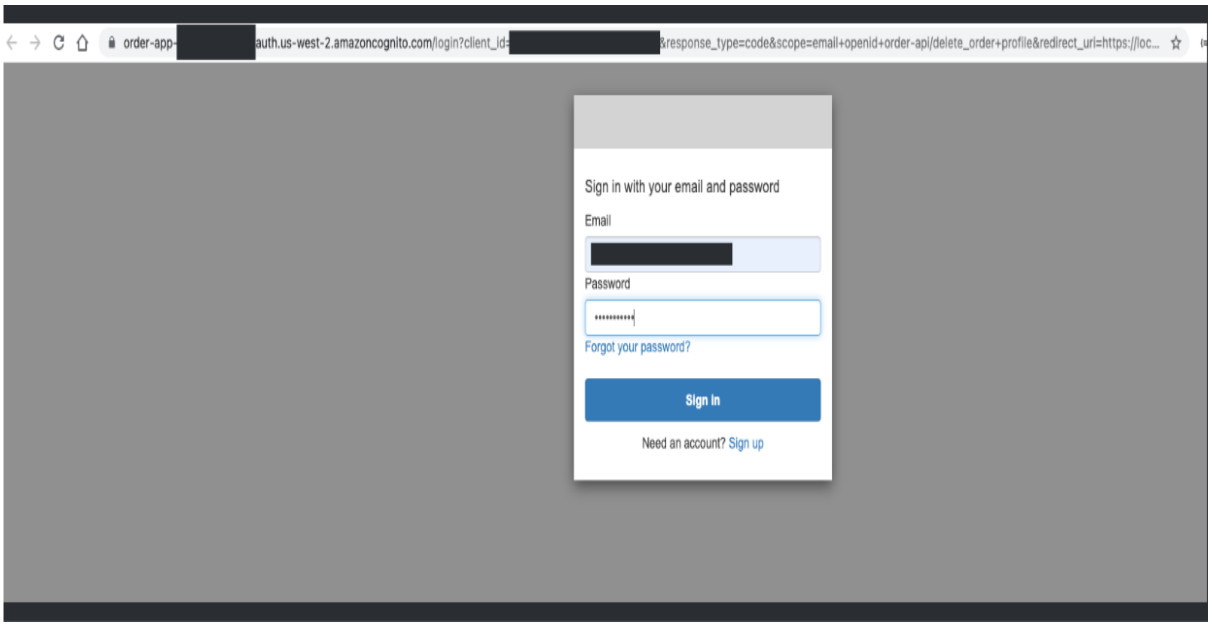
phone [✓ email] [✓ openid] aws.cognito.signin.user.admin [✓ profile]

Allowed Custom Scopes

[✓ order-api/delete_order]

Hosted UI

The hosted UI provides an OAuth 2.0 authorization server with built-in webpages that can be used to sign up and sign in users using the domain you created. Learn more about the hosted UI


order-app-[redacted]auth.us-west-2.amazoncognito.com/login?client_id=[redacted]&response_type=code&scope=email+openid+order-api/delete_order+profile&redirect_uri=https://loc...

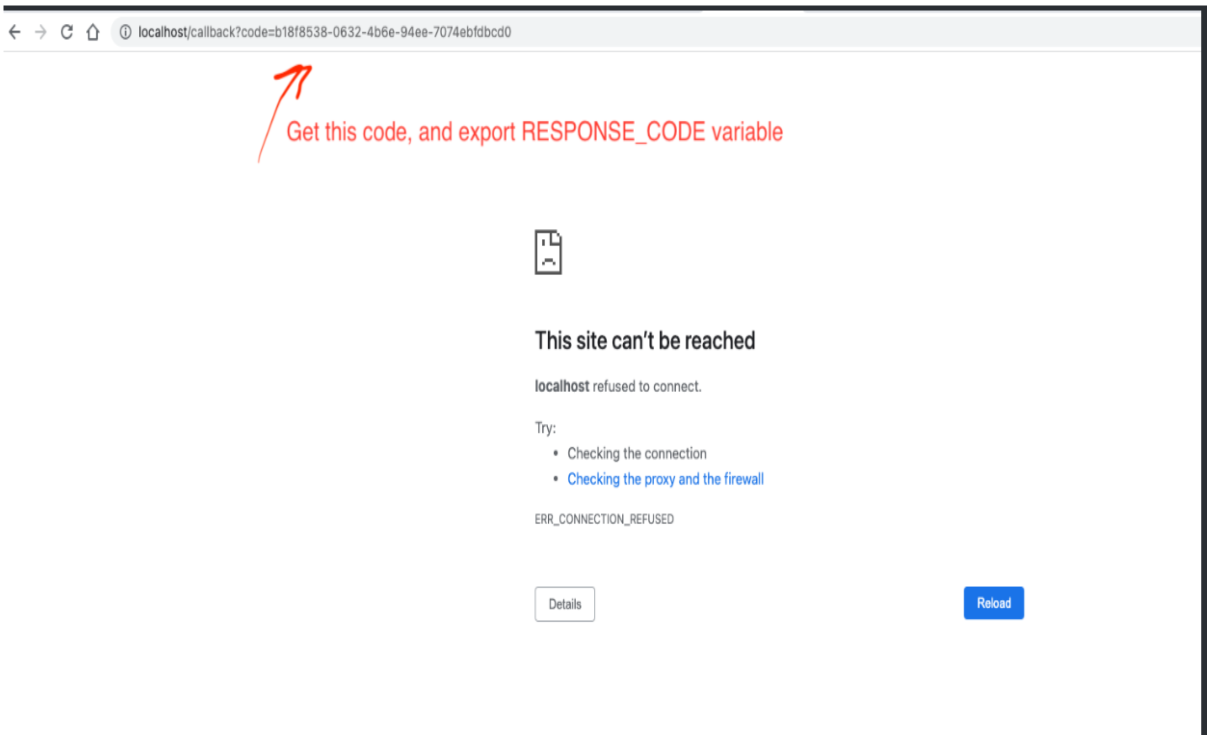
Sign in with your email and password

Email

[redacted]

Password

Forgot your password?


localhost/callback?code=b18f8538-0632-4b6e-94ee-7074ebfdbcd0

Get this code, and export RESPONSE_CODE variable

This site can't be reached

localhost refused to connect.

Try:

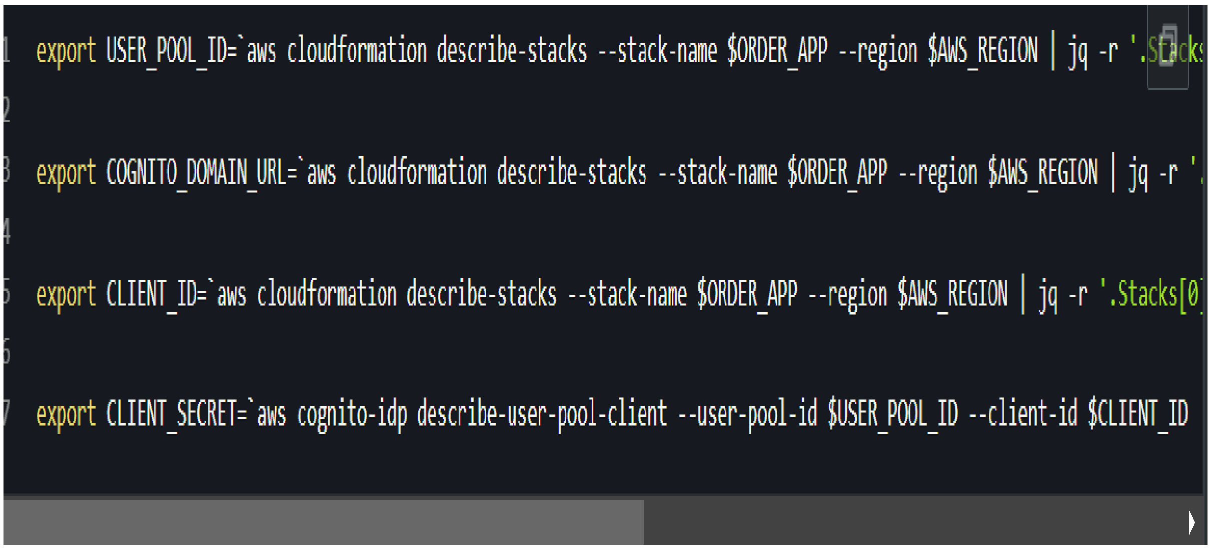
- Checking the connection
- Checking the proxy and the firewall

ERR_CONNECTION_REFUSED

As shown in the screen photo above, get the RESPONSE CODE variable from Cognito Console:

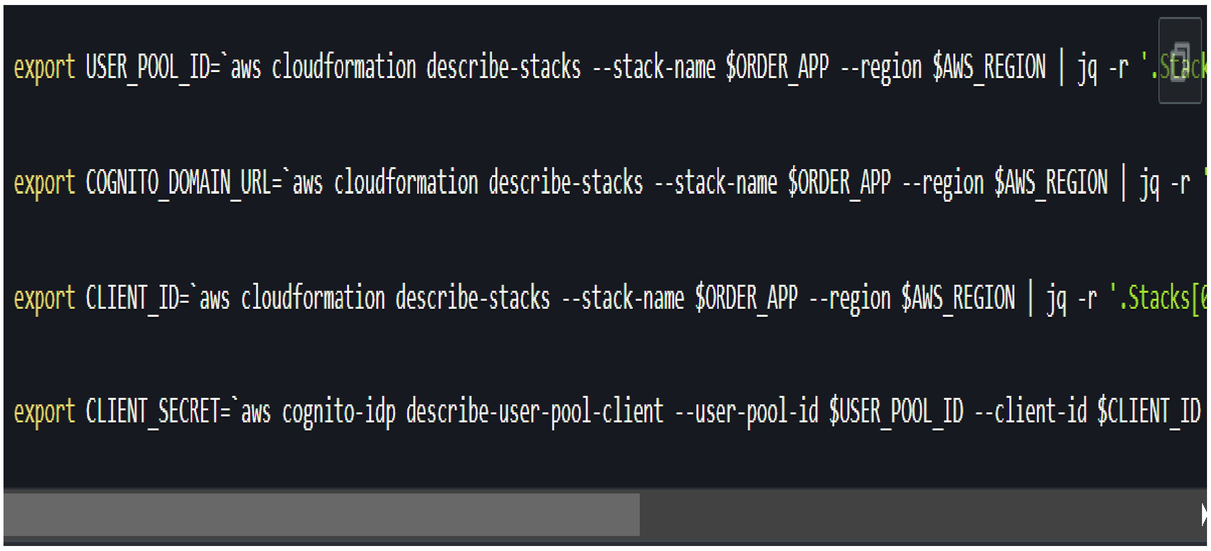
```
export RESPONSE_CODE=<response-code-from-cognito-console>
```

To set the Cognito Domain url, App client id, and App Client secret, run the commands below.



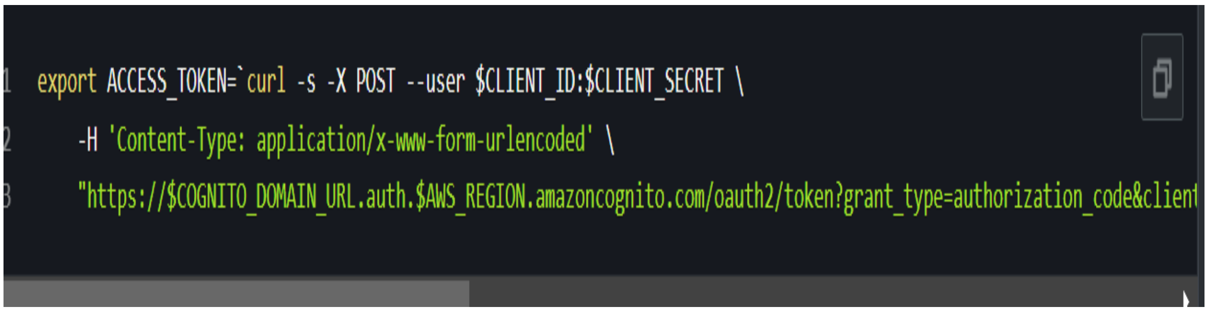
```
export USER_POOL_ID=`aws cloudformation describe-stacks --stack-name $ORDER_APP --region $AWS_REGION | jq -r '.Stacks

export COGNITO_DOMAIN_URL=`aws cloudformation describe-stacks --stack-name $ORDER_APP --region $AWS_REGION | jq -r '.

export CLIENT_ID=`aws cloudformation describe-stacks --stack-name $ORDER_APP --region $AWS_REGION | jq -r '.Stacks[0]

export CLIENT_SECRET=`aws cognito-idp describe-user-pool-client --user-pool-id $USER_POOL_ID --client-id $CLIENT_ID
```

Let's try sending a Curl request to the Cognito Domain to get an Access Token with OAuth2 scope (order-api/delete order), response code, client id, and client secret.



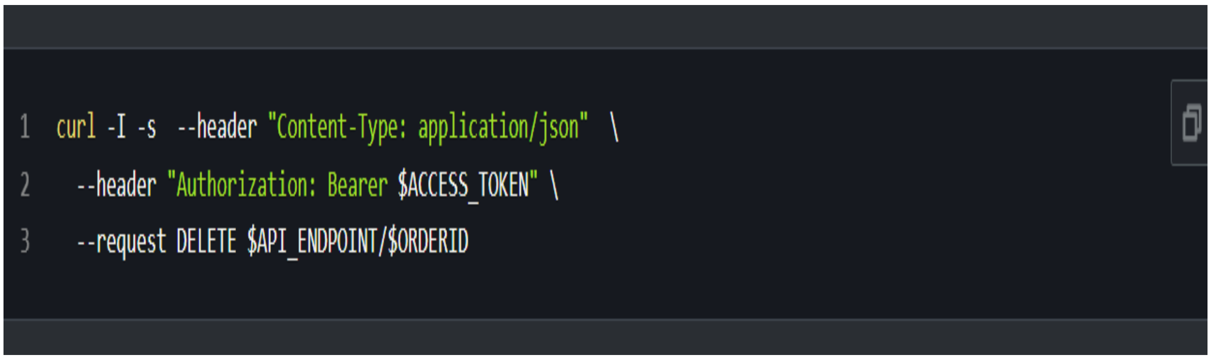
```
export USER_POOL_ID=`aws cloudformation describe-stacks --stack-name $ORDER_APP --region $AWS_REGION | jq -r '.Stack

export COGNITO_DOMAIN_URL=`aws cloudformation describe-stacks --stack-name $ORDER_APP --region $AWS_REGION | jq -r '

export CLIENT_ID=`aws cloudformation describe-stacks --stack-name $ORDER_APP --region $AWS_REGION | jq -r '.Stacks[0

export CLIENT_SECRET=`aws cognito-idp describe-user-pool-client --user-pool-id $USER_POOL_ID --client-id $CLIENT_ID
```

Let's try deleting something using ACCESS TOKEN and ORDERID>.



```
export ACCESS_TOKEN=`curl -s -X POST --user $CLIENT_ID:$CLIENT_SECRET \
   -H 'Content-Type: application/x-www-form-urlencoded' \
   "https://$COGNITO_DOMAIN_URL.auth.$AWS_REGION.amazoncognito.com/oauth2/token?grant_type=authorization_code&client
```

(To list orders and acquire the single order id, go to the Fetch Orders Testing Section.)



```
curl -I -s  --header "Content-Type: application/json"  \
  --header "Authorization: Bearer $ACCESS_TOKEN" \
  --request DELETE $API_ENDPOINT/$ORDERID
```

5.2 Conclusion

In this project we were able to create a serverless application which can use API gateways to fetch the data by making use of the AWS SAM template . The SNS queue was used for order processing and for the purpose of Authentication we made use of Amazon Cognito which provided the results which were intended. The use of AWS SAM template ensures that best practices are used to deploy the application and the system built inturn is scalable. The authentication provides a layer of security to restrict user access as desired by the application vendor, making the communication between client and server secure.

5.3 Application of the project

The order processing mechanism defined within the scope of this project provide its applications in various industries across the market. The API calling for obtaining information can be used to create data abstraction and help in getting important information only. The application used in this project is for food ordering service but it can be further extended to other applications of online transactions like for payment or handling website traffic. The various functions can be used to create , update or delete orders as per the wish of the client by making use of API gateway provided and the order of these requests is maintained by the SNS pipeline used in this project. The authentication method provides an extra layer of security which can be utilised in case of handling of sensitive information.

5.4 Future Work

In Future we can enhance the order management by making use of AWS Step functions to perform various operations like showing payment , order checking and sending notice to the client. The step function can be used in combination with proposed architecture to enhance the client and server interaction. For authentication purposes advanced methods of authentication like OAuth 2.0 service of AWS or API keys can be used to add further layer of security in case of high sensitive data. This can be used if the data that needs obtained to related to banking. This can also provide restricted access to the only user for whom the information is intended.

REFERENCES

1. “Serverless Framework,” *https://serverless.com/,* [Online; accessed 11-September-2017]

2. J. Spillner, “*Snafu: Function-as-a-service (faas) runtime design and implementation*,” CoRR, vol. abs/1703.07562, 2017.

3. “New Relic Monitors Serverless Computing with AWS Lambda Integration,” https://blog.newrelic.com/2016/11/29/ aws-lambda-integration-serverless-infrastructure/, [Online; accessed 11-September-2017].

4. “Dashbird,” *https://dashbird.io/,* [Online; accessed 11- September-2017].

5. “Zipkin,” *http://zipkin.io/,* [Online; accessed 11-September2017].

6. B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, “*Dapper, a large-scale distributed systems tracing infrastructure*,” Google, Inc., Tech. Rep., 2010. [Online]. Available: https: //research.google.com/archive/papers/dapper-2010-1.pdf

7. P. Bailis, A. Fekete, A. Ghodsi, J. Hellerstein, and I. Stoica, “*The potential dangers of causal consistency and an explicit solution*,” in Proceedings of the Third ACM Symposium on Cloud Computing, ser. SoCC ’12, 2012.

8. R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica, “*X-trace: A pervasive network tracing framework*,” in USENIX Conference on Networked Systems Design and Implementation, 2007.

9. R. Escriva, A. Dubey, B. Wong, and E. Sirer, “*Kronos: The design and implementation of an event ordering service*,” in European Conference on Computer Systems, ser. EuroSys ’14, 2014.

10. W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen, “*Don’t settle for eventual: Scalable causal consistency for wide-area storage with cops*,” in ACM Symposium on Operating Systems Principles, 2011.

11. P. Bailis, A. Ghodsi, J. Hellerstein, and I. Stoica, “*Bolt-on causal consistency*,” in ACM SIGMOD International Conference on Management of Data, 2013.

12. M. Bravo, L. Rodrigues, and P. V. Roy, “*Saturn: A distributed metadata service for causal consistency*,” in European Conference on Computer Systems, 2017.

13. https://www.statista.com/topics/5593/digital-payment-in-india/

14. Hazari, S. S., & Mahmoud, Q. H. (2019, January). A parallel proof of work to improve transaction speed and scalability in blockchain systems. In *2019 IEEE 9th annual computing and communication workshop and conference (CCWC)* (pp. 0916-0921). IEEE.

15. Mishra, A. (2017). *Amazon Web Services for Mobile Developers: Building Apps with AWS*. John Wiley & Sons.

16. Yu, J. Y., & Kim, Y. G. (2019, January). Analysis of IoT platform security: A survey. In *2019 International Conference on Platform Technology and Service (PlatCon)* (pp. 1-5). IEEE.

17. Sbarski, P., & Kroonenburg, S. (2017). *Serverless architectures on Aws: with examples using Aws Lambda*. Simon and Schuster.

18. Rajan, R. A. P. (2018, December). Serverless architecture-a revolution in cloud computing. In *2018 Tenth International Conference on Advanced Computing (ICoAC)* (pp. 88-93). IEEE.

19. Lin, W. T., Krintz, C., Wolski, R., Zhang, M., Cai, X., Li, T., & Xu, W. (2018, April). Tracking causal order in aws lambda applications. In *2018 IEEE International Conference on Cloud Engineering (IC2E)* (pp. 50-60). IEEE.

20. zs.com/about/our-story