

Microservices on AWS

Abstract

Microservices are an architectural and organizational approach to software development created to speed up deployment cycles, foster innovation and ownership, improve maintainability and scalability of software applications, and scale organizations delivering software and services by using an agile approach that helps teams work independently. With a microservices approach, software is composed of small services that communicate over well-defined application programming interfaces (APIs) that can be deployed independently. These services are owned by small autonomous teams. This agile approach is key to successfully scale your organization.

Three common patterns have been observed when AWS customers build microservices: API driven, event driven, and data streaming. This whitepaper introduces all three approaches and summarizes the common characteristics of microservices, discusses the main challenges of building microservices, and describes how product teams can use Amazon Web Services (AWS) to overcome these challenges.

Introduction Technical Guide

The technical guide will display how to build an simple CRUD HTTP API Microservice by Lambda, DynamoDB. Then we build a program to manage blogs/comments by integration API and use a service Machine Learning AWS Comprehend to predict, analytics content.

The technical guide include 2 part :

- 1. Part 1: Building A CRUD HTTP API Serverless with Lambda + DynamoDB**
- 2. Part 2: Flask Application – Simple Manage blogs/comments**

Part 1: Building A CRUD HTTP API Serverless with Lambda + DynamoDB

Overviews

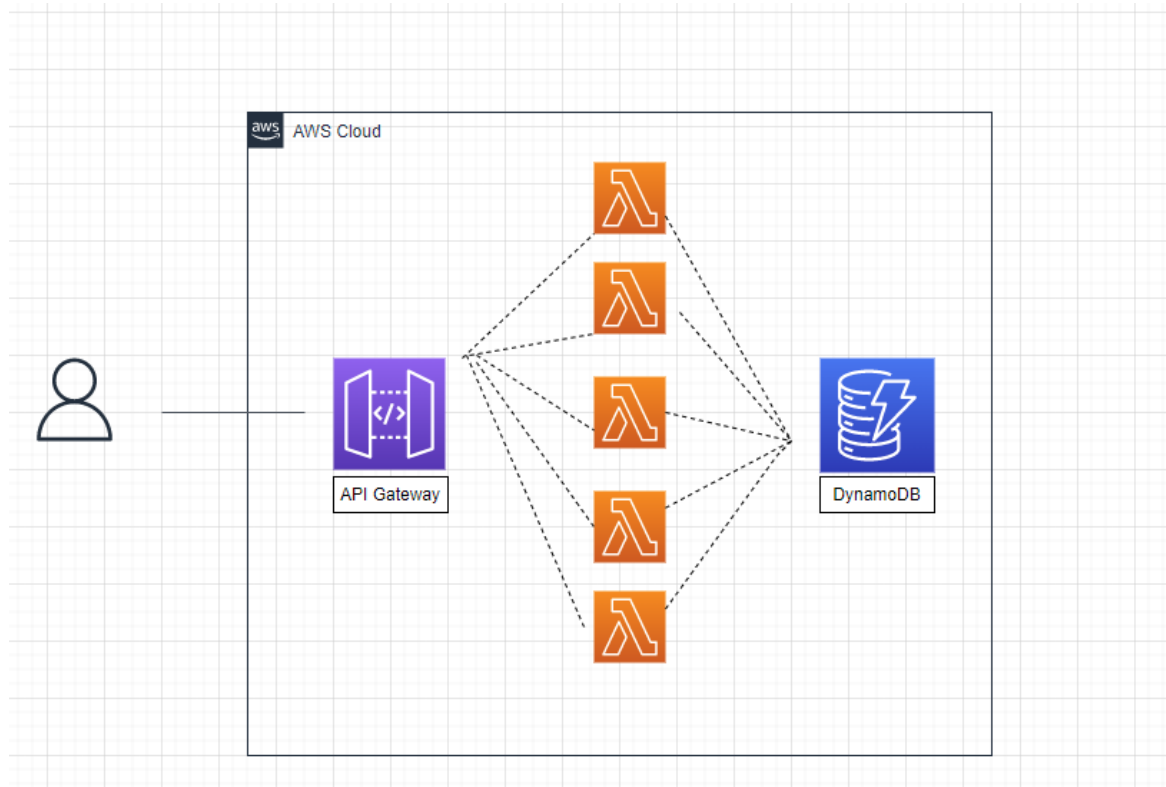
In this article, we are going to build a simple serverless microservice on AWS that enables users to create and manage blog , comment data. We will set up an entire backend using API Gateway, Lambda and DynamoDB. Then we will use Flask Framework to build an application to integrated with Backend and integrate AWS Comprehend , a service machine learning to predict, analytics content .

The goal of this project is to learn how to create an application composed of small, easily deployable, loosely coupled, independently scalable, serverless components.

Architect

1. A user requests to the server by calling APIs from UI. User's request which includes all necessary information is sent to Amazon API Gateway restful service.
2. API Gateway transfers the collected user information to a particular AWS Lambda function based on user's request.
3. AWS Lambda function executes event-based logic calling DynamoDB database.
4. DynamoDB provides a persistence layer where data can be stored/retrieved by the API's Lambda function.

The high-level architecture for the serverless microservice is illustrated in the diagram below:



AWS Resource

- AWS API Gateway
- AWS Lambda
- AWS DynamoDB
- AWS IAM
- AWS SAM
- AWS SDK Python Boto3

Implementation CRUD HTTP API Serverless

HTTP APIs are designed for low-latency, cost-effective integrations with AWS services, including AWS Lambda, and HTTP endpoints. HTTP APIs support OIDC and OAuth 2.0 authorization, and come with built-in support for CORS and automatic deployments. Previous-generation REST APIs currently offer more features.

Topics

- Step 1: Create a DynamoDB table
- Step 2: Create a Lambda function
- Step 3: Create an HTTP API
- Step 4: Create routes
- Step 5: Create an integration
- Step 6: Attach your integration to routes
- Step 7: Test your API
- Step 8: Clean up

Step 1 : Create A DynamoDB table

You use a **DynamoDB** table to store data for your API.

Each item has a unique ID, which we use as the **partition key** for the table.

To create a DynamoDB table

1. Open **DynamoDB** console at <https://console.aws.amazon.com/dynamodb/>
2. Choose **Create table**
3. For table name, enter <table_name> (ex: tbl_comment)
4. For **Primary key**, enter **uuid**

5. Choose **Create**

Create table

Table details [Info](#)
DynamoDB is a schemaless database that requires only a table name and a primary key when you create the table.

Table name
This will be used to identify your table.

Between 3 and 255 characters, containing only letters, numbers, underscores (_), hyphens (-), and periods (.).

Partition key
The partition key is part of the table's primary key. It is a hash value that is used to retrieve items from your table and allocate data across hosts for scalability and availability.

1 to 255 characters and case sensitive.

Sort key - optional
You can use a sort key as the second part of a table's primary key. The sort key allows you to sort or search among all items sharing the same partition key.

1 to 255 characters and case sensitive.

Settings

☒ **Default settings**
The fastest way to create your table. You can modify these settings now or after your table has been created.

☐ **Customize settings**
Use these advanced features to make DynamoDB work better for your needs.

Step 2: Create a Lambda function

You create a Lambda function for the backend of your API. This Lambda function creates, reads, updates, and deletes items from DynamoDB. The function uses events from API Gateway to determine how to interact with DynamoDB. As a best practice, you should create separate functions for each route.

We will create **5 Lambda function** :

- **Create Item** blog/comment function(create_item/lambda_function.py)
 - Trigger by POST /items API
- **Get item** blog/comment function (get_item/lambda_function.py)
 - Trigger by GET /items/{uuid} API
- **Get all items** blog/comment function (getAlls/lambda_function.py)
 - Trigger by GET /items API
- **Update item** blog/comment function (update_item/lambda_function.py)
 - Trigger by PUT /items API
- **Delete item** blog/comment function (delete_item/lambda_function.py)
 - Trigger by DELETE /items/{uuid} API

Let's start with Get item

1. **Sign in** to the Lambda console at <https://console.aws.amazon.com/lambda>
2. Choose **Create Function**:
 - a. We have 4 options to create lambda function. In this lab, we choose option 1.
3. Choose **Author from scratch**
4. For **Function name**, enter <function name> (ex : get_item)
5. **Runtime** : Python 3.x
6. Under **Permissions** choose **Change default execution role**
7. Select **Create a new role from AWS policy templates**
8. For **Role name**, enter <get_item_role>
9. For **Policy templates**, choose Simple microservice permissions. This policy grants the Lambda function permission to interact with DynamoDB.
 - a. ***Note:**
This tutorial uses a managed policy for simplicity. As a best practice, you should create your own IAM policy to grant the minimum permissions required.
10. Choose **Create function**.
11. Open Lambda_function, **edit code**.
12. Choose **Deploy** to update your function.

The screenshot shows the AWS Lambda 'Create function' page. The 'Author from scratch' tab is active. The 'Function name' field contains 'get_item'. The 'Runtime' is set to 'Python 3.8'. The 'Architecture' is 'x86_64'. In the 'Permissions' section, 'Change default execution role' is selected. Under 'Execution role', 'Create a new role from AWS policy templates' is chosen. A message states: 'Role creation might take a few minutes. Please do not delete the role or edit the trust or permissions policies in this role.' The 'Role name' field contains 'get_item_role'. In the 'Policy templates - optional' dropdown, 'Simple microservice permissions' is selected, and a tag indicates it includes 'DynamoDB'.

In the same way as above, we need to create four more Lambda functions:

- Create item function, get code from Github
- Get all items function, get code from Github
- Update item function, get code from Github
- Delete item function, get code from Github

Lambda > Functions

Functions (7) Last fetched 20 seconds ago Actions Create function

Filter by tags and attributes or search by keyword

| <input type="checkbox"/> | Function name | Description | Package type | Runtime | Code size | Last modified |
|--------------------------|---------------|-------------|--------------|------------|------------|---------------|
| <input type="checkbox"/> | create_item | - | Zip | Python 3.8 | 607.0 byte | 5 hours ago |
| <input type="checkbox"/> | insert_db | - | Zip | Python 3.8 | 536.0 byte | 2 days ago |
| <input type="checkbox"/> | getAlls | - | Zip | Python 3.8 | 544.0 byte | 2 days ago |
| <input type="checkbox"/> | create_db | - | Zip | Python 3.8 | 892.0 byte | 5 hours ago |
| <input type="checkbox"/> | delete_item | - | Zip | Python 3.8 | 686.0 byte | 2 days ago |
| <input type="checkbox"/> | get_item | - | Zip | Python 3.8 | 479.0 byte | 1 day ago |
| <input type="checkbox"/> | update_item | - | Zip | Python 3.8 | 560.0 byte | 2 days ago |

Step 3: Create an HTTP API

The HTTP API provides an HTTP endpoint for your Lambda function. In this step, you create an empty API. In the following steps, you configure routes and integrations to connect your API and your Lambda function.

To create an HTTP API

1. Sign in to the **API Gateway** console at <https://console.aws.amazon.com/apigateway>
2. Choose **Create API**, and the for **HTTP API**, choose **Build**.
3. For **API name**, enter <api_gateway_name> (ex : http-crud-tutorial-item-blog-api)
4. Choose **Next**.
5. For **Configure routes**, choose **Next** to skip route creation. You create routes later.
6. Review the stage that API Gateway creates , and the choose **Next**.
7. Choose **Create**

The screenshot shows the AWS API Gateway console. At the top, there's a breadcrumb 'API Gateway > Details' and a 'Deploy' button. The API name 'http-crud-tutorial-item-blog-api' is displayed with an 'Edit' button. Below this is the 'API details' section, which includes a table with the following information:

| API ID | Protocol | Created |
|----------------|------------------|------------|
| s1kdb8ov6j | HTTP | 2022-03-15 |
| Description | Default endpoint | |
| No Description | Enabled | |

Below the API details is the 'Stages for http-crud-tutorial-item-blog-api' section. It contains a search bar and a table with the following data:

| Stage name | Invoke URL | Attached deployment | Auto deploy | Last updated |
|------------|---|---------------------|-------------|--------------|
| \$default | https://s1kdb8ov6j.execute-api.ap-southeast-1.amazonaws.com | 3ggrdp | enabled | 2022-03-17 |

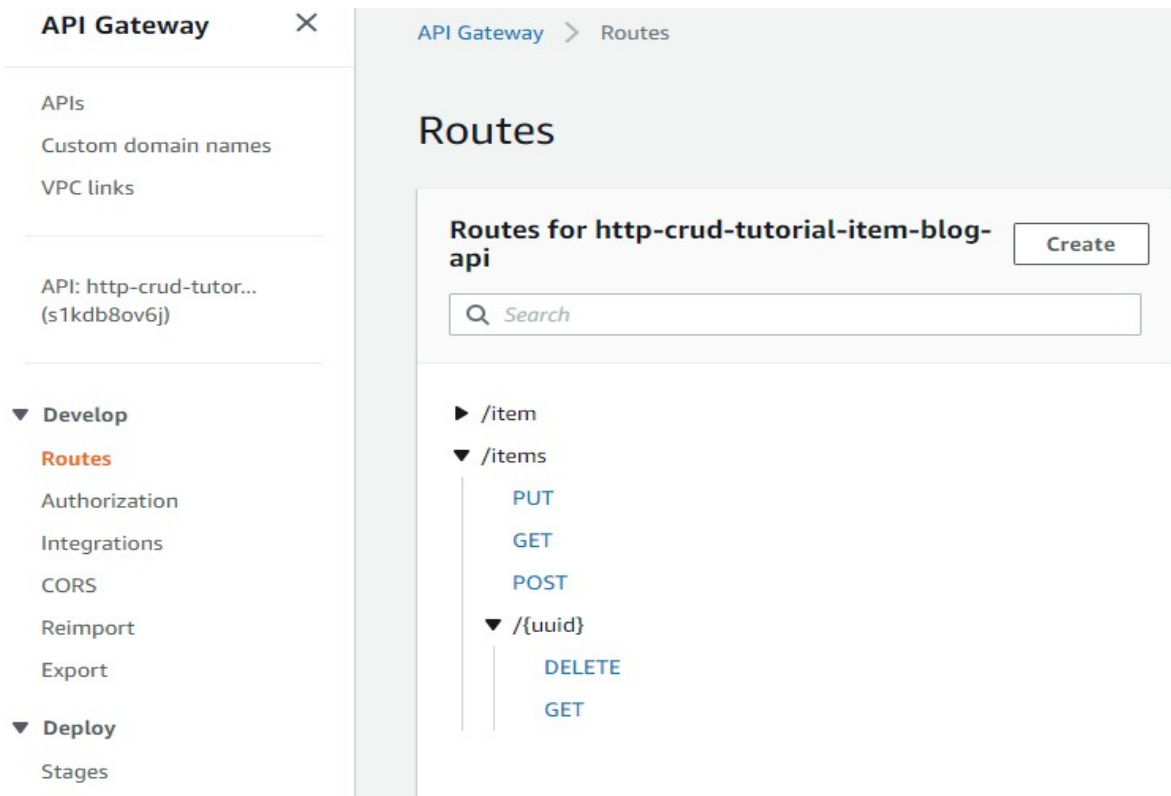
Step 4: Create routes

Routes are a way to send incoming API requests to backend resources. Routes consist of two parts: an HTTP method and a resource path, for example, GET /items. For this example API, we create five routes:

- GET /items/{uuid}
- GET /items
- PUT /items
- DELETE /items/{uuid}
- POST /items

To create routes

1. Sign in to the API Gateway console at <https://console.aws.amazon.com/apigateway>
2. Choose your **API**
3. Choose **Routes**
4. Choose **Create**
5. For **Method**, choose **GET**.
6. For the path, enter **/items/{uuid}**. The **{uuid}** at the end of path is path parameters that API Gateway retrieves from the request path when a client makes a request.
7. Choose **Create**.
8. Repeat steps 4-7 for **POST /items**, **GET /items**, **DELETE /items/{uuid}**, and **PUT /items**.



Step 5: Create an integration

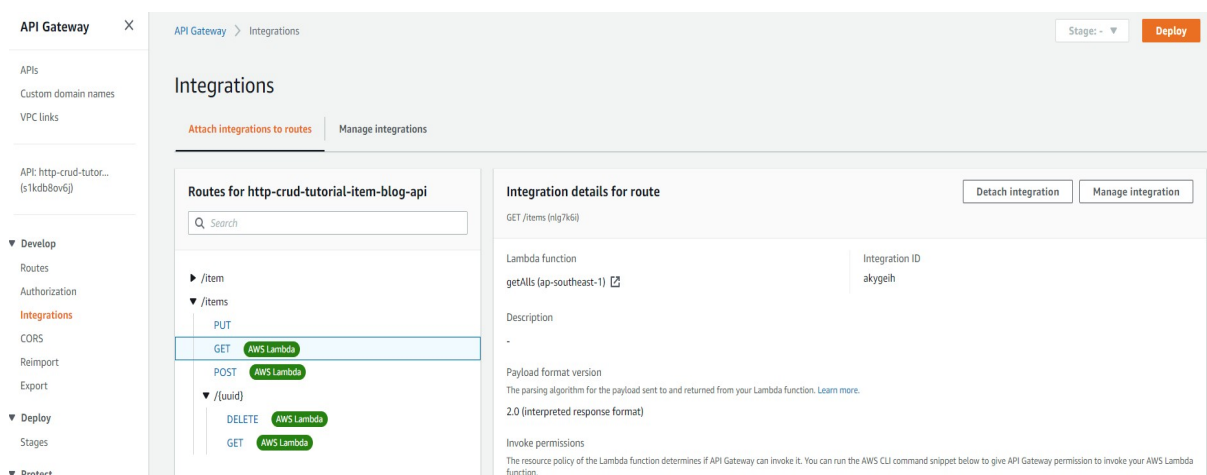
You create an integration to connect a route to backend resources. For this example API, you create one Lambda integration that you use for all routes.

To create an integration

1. Sign in to the API Gateway console at <https://console.aws.amazon.com/apigateway>
2. Choose your **API**.
3. Choose **Integration**.
4. Choose **Manage integrations** and then choose **Create**.
5. Skip **Attach this integration to a route**. You complete that in a later step
6. For **Integration type**, choose **Lambda function**
7. For **Lambda function**, enter <function name> (ex : get_item)
8. Choose **Create**.

Step 6: Attach your integration to routes

1. Sign in to the API Gateway console at <https://console.aws.amazon.com/apigateway>.
2. Choose your API.
3. Choose **Integrations**.
4. Choose a route.
5. Under **Choose an existing integration**, choose enter <function name> (ex : get_item)
6. Choose **Attach integration**.
7. Repeat steps 4-6 for all routes.



Step 7: Test your API

To make sure that your API is working, you use add-on **REST API** or **curl**.

To get the URL to invoke your API

1. Sign in to the API Gateway console at <https://console.aws.amazon.com/apigateway>.
2. Choose your API.
3. Note your API's invoke URL. It appears under **Invoke URL** on the **Details** page.

The screenshot shows the AWS API Gateway console. On the left is a navigation menu with options like 'APIs', 'Custom domain names', 'VPC links', 'Develop', 'Deploy', and 'Protect'. The main panel displays the details for the API 'http-crud-tutorial-item-blog-api'. It includes an 'API details' section with fields for API ID, Protocol, Created, Description, and Default endpoint. Below this is a 'Stages for http-crud-tutorial-item-blog-api' section with a table listing stages, their invoke URLs, attached deployments, auto-deploy status, and last updated times. The 'default' stage is listed with an invoke URL that includes a redacted region name and the endpoint 'execute-api.ap-southeast-1.amazonaws.com'. There are also 'Tags (0)' at the bottom.

To get all items

The full URL looks like `https://xxxxx.execute-api.<region-name>.amazonaws.com/items`

The screenshot shows a web browser with a REST client interface. The URL bar shows the endpoint `https://xxxxx.execute-api.ap-southeast-1.amazonaws.com/items`. The response is a JSON array of three items. Each item contains a 'content' field with a paragraph of text, a 'uuid' field, and a 'title' field. The titles are 'Your AnyCompany Financial Service', 'Truman Show', and 'United States_2'.

To get an item

The full URL looks like `https://xxxxx.execute-api.<region-name>.amazonaws.com/items/36759243-a5cb-11ec-9612-2fd7bd8a6312`

The screenshot shows a web browser with a REST client interface. The URL bar shows the endpoint `https://xxxxx.execute-api.ap-southeast-1.amazonaws.com/items/36759243-a5cb-11ec-9612-2fd7bd8a6312`. The response is a JSON object representing a single item. It includes a 'content' field, a 'uuid' field, a 'title' field, and a 'ResponseMetadata' object containing 'RequestId', 'HTTPStatusCode', 'HTTPHeaders', and 'RetryAttempts'.

To add an item blog/comment

The full URL looks like `https://xxxxx.execute-api.<region-name>.amazonaws.com/items`
json input :

```
{
  "event": {
    "httpMethod": "POST",
    "body": {
      "title": "Great video",
      "content": "Great video and I love Ben's approach to MLE. What specific frameworks and tools do you recommend using for our MLE work? Do you have any favorites that help implement SWE best practices? What about design patterns for ML?"
    }
  }
}
```

output

↗ SureUtils » REST API Client

Request

URL

https://[redacted].execute-api.ap-southeast-1.amazonaws.com/items

Method

GET POST PUT DELETE HEAD OPTIONS

Headers

Encoding

UTF-8

Content-Type

application/json

Additional Headers

Add

Data

```
{
  "event": {
    "httpMethod": "POST",
    "body": {
      "title": "Great video",
      "content": "Great video and I love Ben's approach to MLE. What specific frameworks and tools do you recommend using for our MLE work? Do you have any favorites that help implement SWE best practices? What about design patterns for ML?"
    }
  }
}
```

Clear Submit

Response

Status

200 OK

Headers

```
apigw-requestid: PHtcBjdSyQ0EIH4=
content-length: 61
content-type: text/plain; charset=utf-8
date: Thu, 17 Mar 2022 08:48:40 GMT
```

Data

```
{"message": "A new item was saved successfully in database"}
```

Step 8: Clean up

To prevent unnecessary costs, delete the resources that you created as part of this getting started exercise. The following steps delete your HTTP API, your Lambda function, and associated resources.

To delete a DynamoDB table

1. Open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. Select your table.
3. Choose **Delete table**.
4. Confirm your choice, and choose **Delete**.

To delete an HTTP API

1. Sign in to the API Gateway console at <https://console.aws.amazon.com/apigateway>.
2. On the **APIs** page, select an API. Choose **Actions**, and then choose **Delete**.
3. Choose **Delete**.

To delete a Lambda function

1. Sign in to the Lambda console at <https://console.aws.amazon.com/lambda>.
2. On the **Functions** page, select a function. Choose **Actions**, and then choose **Delete**.
3. Choose **Delete**.

To delete a Lambda function's log group

1. In the Amazon CloudWatch console, open the [Log groups page](#).
2. On the **Log groups** page, select the function's log group (`/aws/lambda/get-item`). Choose **Actions**, and then choose **Delete log group**.
3. Choose **Delete**.

To delete a Lambda function's execution role

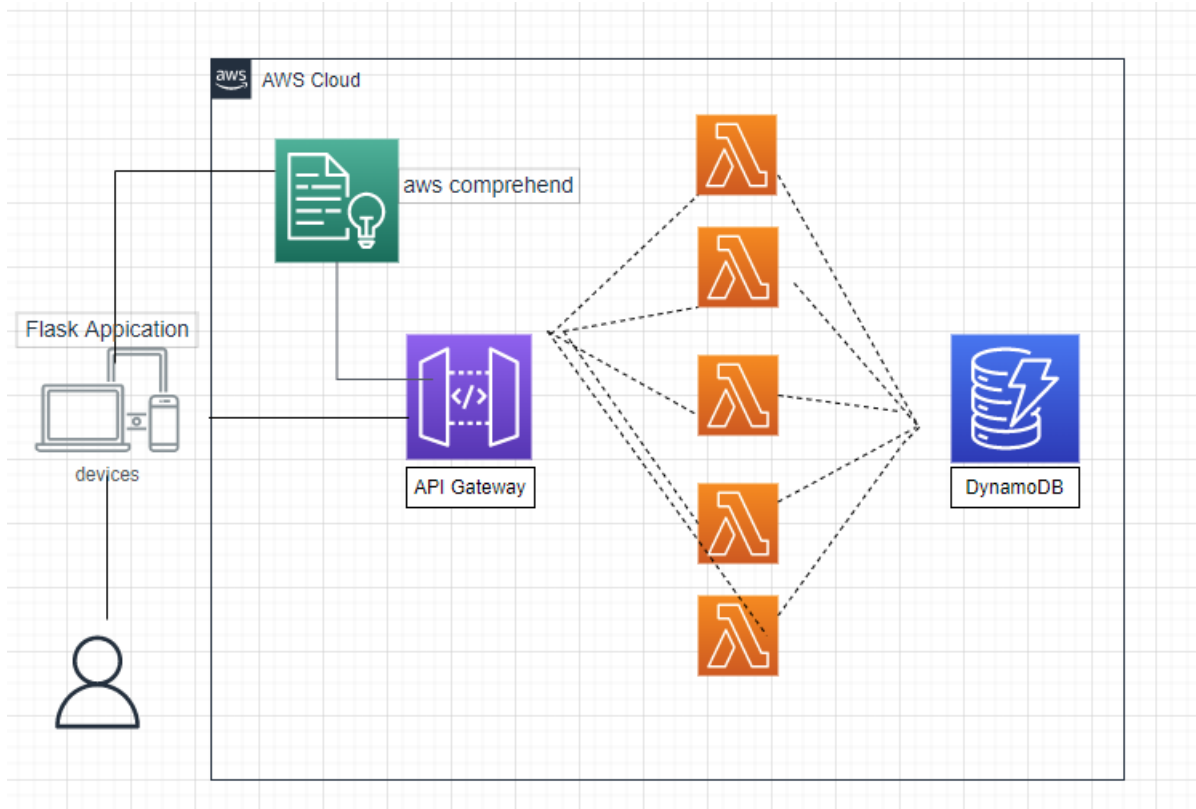
1. In the AWS Identity and Access Management console, open the [Roles page](#).
2. Select the function's role, for example, `get-item-role`.
3. Choose **Delete role**.
4. Choose **Yes, delete**.

Part 2 : Demo Application

Overviews

We build an application to integration with API to display content on UI. Then we use an service Machine Learning AWS Comprehend to predict, analytics content.

Architect



Technical Application:

Flask Application

- IDE : Pycharm
- Language : Python 3.9
- Deloyment : Localhost
- Feature :

```
@app.route('/getAlls',methods = ['GET'])
@app.route('/predict/<uuid>',methods = ['GET'])
@app.route('/items/<uuid>',methods = ['GET'])
```

Integrated with AWS Comprehend

Amazon Comprehend uses natural language processing (NLP) to extract insights about the content of documents. Amazon Comprehend processes any text file in UTF-8 format, image files (JPG, PNG, or TIFF), and semi-structured

documents (PDF or Word files). It develops insights by recognizing the entities, key phrases, language, sentiments, and other common elements in a document. Use Amazon Comprehend to create new products based on understanding the structure of documents. For example, using Amazon Comprehend you can search social networking feeds for mentions of products or scan an entire document repository for key phrases.

Implementation

For demo, we developed 3 functions : Get all items blog/comment, get an item , and predict by AWS Comprehend.

```
@app.route('/')
def hello_world():
    return "Hello World"
@app.route('/getALLs', methods = ['GET'])
def getALLs():
    url = "https://s1.amazonaws.com/execute-api.ap-southeast-1.amazonaws.com/items"
    res = requests.get(url)
    data = res.json()
    # return jsonify(data)
    return render_template('blogs.html', data=data)

@app.route('/items/<uuid>', methods = ['GET'])
def getItem(uuid):
    url = "https://s1.amazonaws.com/execute-api.ap-southeast-1.amazonaws.com/items/" + uuid
    res = requests.get(url)
    data = res.json()
    post = data["Item"]
    return render_template('content.html', post=post)
@app.route('/predict/<uuid>', methods = ['GET'])
def predict(uuid):
    url = "https://s1.amazonaws.com/execute-api.ap-southeast-1.amazonaws.com/items/" + uuid
    res = requests.get(url)
    data = res.json()
    post = data["Item"]
    content = post['content']
    entities, phrases, pii_entities, sentiment = usage_demo(content)
    return jsonify(entities, pii_entities, phrases, sentiment)
```

```
blogpost C:\Users\vnhanh\PycharmProjects\134
> templates
> venv library root
app.py
comprehend_detect.py
> External Libraries
Scratches and Consoles

135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173

def usage_demo(content):
    print("Welcome to the Amazon Comprehend detection demo!")
    session = boto3.Session(
        aws_access_key_id='AKIAVBZIJR7KNENWLSKV',
        aws_secret_access_key='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',
        region_name='ap-southeast-1'
    )
    comp_detect = ComprehendDetect(session.client('comprehend'))
    demo_size = 3
    sample_text = content
    print("Detecting languages.")
    languages = comp_detect.detect_languages(sample_text)
    lang_code = languages[0]['LanguageCode']
    print("Detecting entities.")
    entities = comp_detect.detect_entities(sample_text, lang_code)
    print(f"The first {demo_size} are:")

    print("Detecting key phrases.")
    phrases = comp_detect.detect_key_phrases(sample_text, lang_code)
    print(f"The first {demo_size} are:")
    pprint(phrases[:demo_size])

    print("Detecting personally identifiable information (PII).")
    pii_entities = comp_detect.detect_pii(sample_text, lang_code)
    print(f"The first {demo_size} are:")
    pprint(pii_entities[:demo_size])

    print("Detecting sentiment.")
    sentiment = comp_detect.detect_sentiment(sample_text, lang_code)
    print(f"Sentiment: {sentiment['Sentiment']}")
    print("SentimentScore:")
    pprint(sentiment['SentimentScore'])

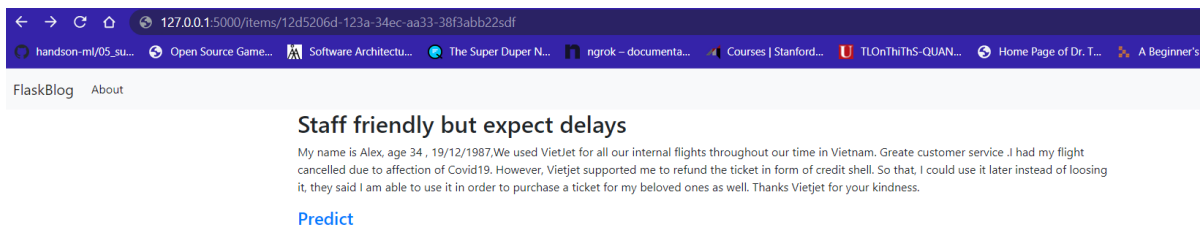
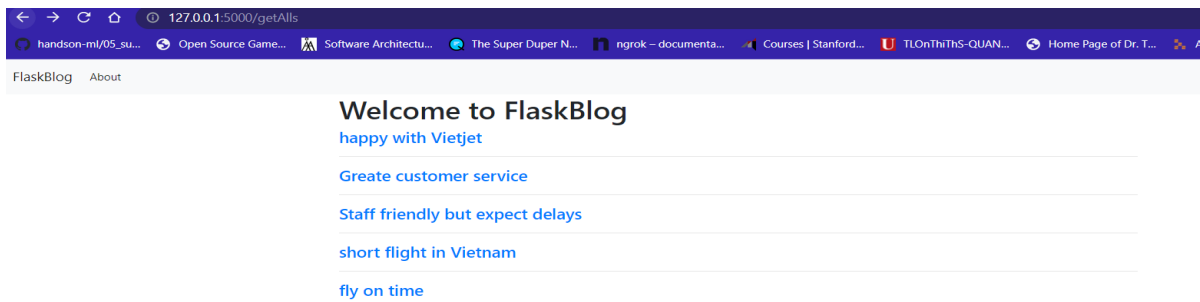
    print("Detecting syntax elements.")
    syntax_tokens = comp_detect.detect_syntax(sample_text, lang_code)
    print(f"The first {demo_size} are:")
    pprint(syntax_tokens[:demo_size])

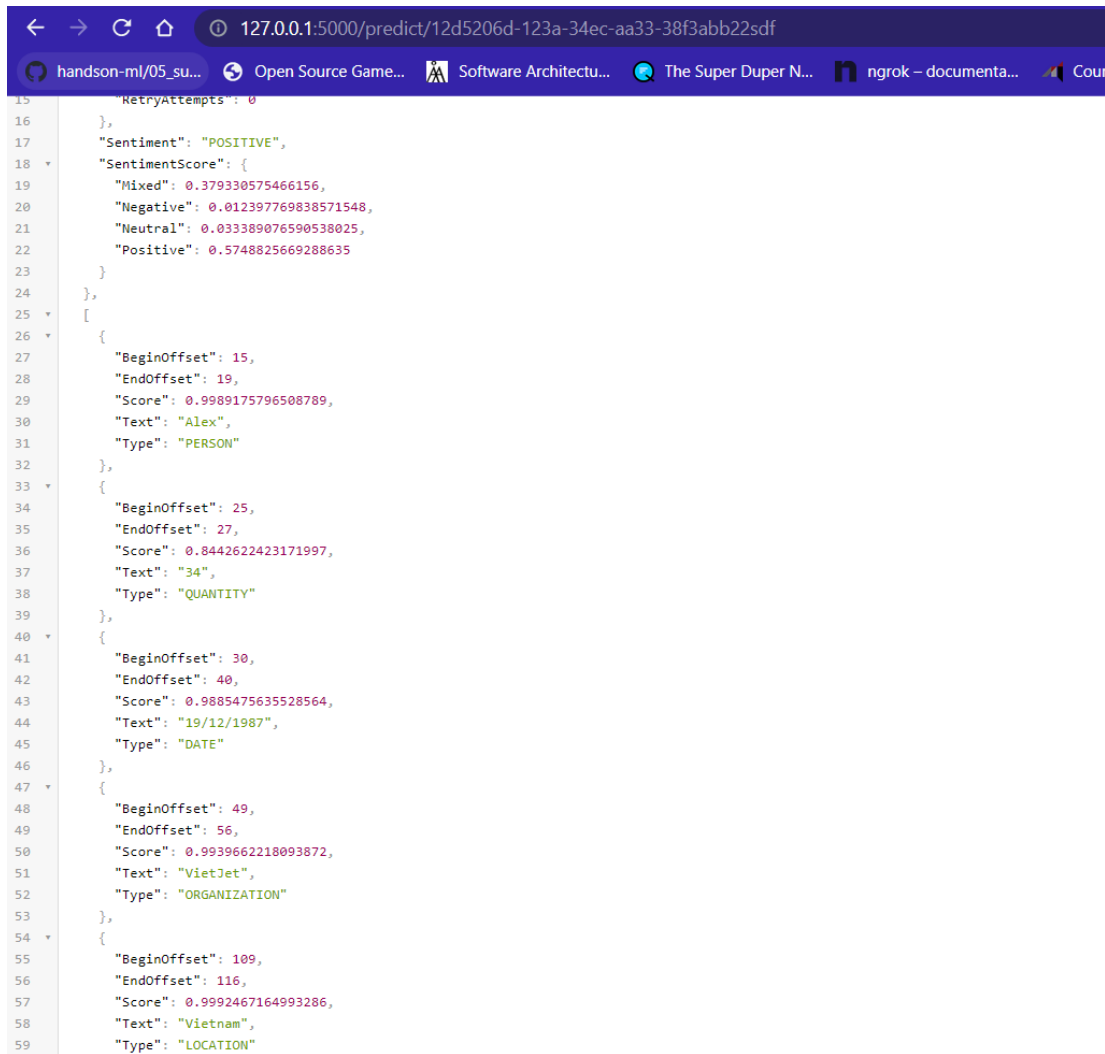
    return entities, phrases, pii_entities, sentiment
```

Some Screen Results of Application

The flow we will test with application:

- Get all items -> Display 1 item -> Predict, analytics this item





```

15     "RetryAttempts": 0
16   },
17   "Sentiment": "POSITIVE",
18   "SentimentScore": {
19     "Mixed": 0.379330575466156,
20     "Negative": 0.012397769838571548,
21     "Neutral": 0.033389076590538025,
22     "Positive": 0.5748825669288635
23   }
24 },
25 [
26   {
27     "BeginOffset": 15,
28     "EndOffset": 19,
29     "Score": 0.9989175796508789,
30     "Text": "Alex",
31     "Type": "PERSON"
32   },
33   {
34     "BeginOffset": 25,
35     "EndOffset": 27,
36     "Score": 0.8442622423171997,
37     "Text": "34",
38     "Type": "QUANTITY"
39   },
40   {
41     "BeginOffset": 30,
42     "EndOffset": 40,
43     "Score": 0.9885475635528564,
44     "Text": "19/12/1987",
45     "Type": "DATE"
46   },
47   {
48     "BeginOffset": 49,
49     "EndOffset": 56,
50     "Score": 0.9939662218093872,
51     "Text": "VietJet",
52     "Type": "ORGANIZATION"
53   },
54   {
55     "BeginOffset": 109,
56     "EndOffset": 116,
57     "Score": 0.9992467164993286,
58     "Text": "Vietnam",
59     "Type": "LOCATION"

```

Explain more Predict :

With text “My name is Alex, age 34 , 19/12/1987,We used VietJet for all our internal flights throughout our time in Vietnam.Greate customer service .I had my flight cancelled due to affection of Covid19.However, Vietjet supported me to refund the ticket in form of credit shell.So that, I could use it later instead of loosing it, they said I am able to use it in order to purchase a ticket for my beloved ones as well.
Thanks Vietjet for your kindness.”

Model Machine Learing Predict, Analytics :

- Sentiment : **Positive**
- Enti Organization : **Vietjet** (Brand)
- Enti Person : **Alex**
- Entity Location : **Vietnam**

and more ...

Summary

You know how to create a simple CRUD (create, read, update, delete) app, and to set the foundational services, components, and plumbing needed to get a basic AWS serverless microservice up and running.

Challenge

- Use CloudFormation to deploy.
- Complete Flask app and public to ec2 or somewhere on the earth.
- A Feature attach image or photo of blog/comment
- Integration more AWS machine Learning, such as AWS Rekognition to detect Brand/Logo Vietjet, HDBank ...
- Setup pipeline CI/CD Application.