



AWS Serverless API Creation Utilities

Introduction

The utilities provided are intended to simplify the process of defining a serverless REST API using minimal JSON definitions. These utilities are implemented as Python scripts and assume a Python environment (virtual or otherwise) that meets the following conditions:

- A 3.x Python interpreter is in the path (scripts were tested with 3.7 and 3.10)
- The utility scripts are stored in an appropriate location relative to the PYTHONPATH environment variable
- The Python argparse library has been installed and is available as an import
- The AWS boto3 library has been installed and is available as an import
- The executing user and machine have been authenticated in the target AWS account (via temporary credentials or otherwise)

The architecture is based on AWS services for IAM, CloudWatch and API Gateway, implemented via Lambda execution accessing data in AWS (S3, DynamoDB, Athena/Glue and/or Redshift). The utility for Lambda function creation assumes that the Lambda code to implement each method has been deployed to an S3 location as a .zip file.

The JSON used as input to these utilities is easily generated, but it can also be created/modified manually. The sample JSON files (provided in the distribution .zip and used in this document) provide simple examples to illustrate the expected JSON structure. The command-line utility scripts are distributed as source code and expected to be enhanced, combined, and/or modified to meet the specific needs of the implementer.

IAM Policies and Roles

First, any API implemented by Lambda will need execution roles (and attached policies) that will allow it to access data and do simple CloudWatch logging. The policies and/or roles in this example may already exist. They are provided here to illustrate the policies and roles needed for Lambda execution accessing data in AWS.

First, the implementation assumes that all Lambdas will do basic CloudWatch logging and will have a policy that can be defined in JSON as:

```
{
  "Name": "MyAPI_lambda_logging_policy",
  "PolicyDoc": {
    "Version": "2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Action": "logs:CreateLogGroup",
        "Resource": "arn:aws:logs:<region>:<account>:*"
      },
      {
        "Effect": "Allow",
```



```
        "Action": [
            "logs:CreateLogStream",
            "logs:PutLogEvents"
        ],
        "Resource": "arn:aws:logs:<region>:<account>:log-group:/aws/lambda/*"
    }
}
}
```

The policy utility will operate on any JSON document that contains a policy name and an embedded policy document, as illustrated below (assuming that it is in a file named the same as the policy with a .json extension):

```
python api_iam_policy_creator.py --acct <acct_num> --file <my-local-
location>/MyAPI_lambda_logging_policy.json
```

Any Lambda designed to present and/or manipulate data will also need appropriate permissions on the underlying data. This example (probably more open than your organization would permit) allows read/write access to all data stored in S3 and all data accessible to Athena/Glue:

```
{
  "Name": "MyAPI_data_access_policy",
  "PolicyDoc": {
    "Version": "2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Action": [
          "athena:*"
        ],
        "Resource": [
          "*"
        ]
      },
      {
        "Effect": "Allow",
        "Action": [
          "glue:*"
        ],
        "Resource": [
          "*"
        ]
      },
      {
        "Effect": "Allow",
        "Action": [
          "s3:*"
        ],
        "Resource": [
          "*"
        ]
      }
    ]
  }
}
```



The details of the data access policy document will differ depending on the data source, but the policy is created using the same utility:

```
python api_iam_policy_creator.py --acct <acct_num> --file <my-local-  
location>/MyAPI_data_access_policy.json
```

The execution role for the implementing Lambda should have these policies attached and can be defined in JSON as:

```
{  
  "Name": "MyAPI_Main_API_role",  
  "AssumeRoleDoc": {  
    "Version": "2012-10-17",  
    "Statement": [  
      {  
        "Effect": "Allow",  
        "Principal": {  
          "Service": "lambda.amazonaws.com"  
        },  
        "Action": "sts:AssumeRole"  
      }  
    ]  
  },  
  "Policies": [  
    "arn:aws:iam::<account>:policy/MyAPI_data_access_policy",  
    "arn:aws:iam::<account>:policy/MyAPI_lambda_logging_policy"  
  ]  
}
```

If this JSON is stored in a file named after the role (with a .json extension), it can be created in IAM with the following command:

```
python api_iam_role_creator.py --acct <acct_num> --file <my-local-  
location>/MyAPI_Main_API_role.json
```

The “AssumeRoleDoc” inline policy allows the Lambda service to assume this role. This role should be used as the execution role when creating the Lambda function itself.

Lambda Functions

The code for the Lambda function to access the data is assumed to be complete and staged in an S3 location as a .zip file. Based on these assumptions, a Lambda to perform a search by name may be created using the JSON below:

```
{  
  "FunctionName": "MyAPI_get_by_name_like",  
  "ExecutionRole": "arn:aws:iam::<account>:role/ MyAPI_Main_API_role",  
  "DeploymentBucket": "<my-s3-bucket-name>",  
  "DeploymentKey": "<my-lambda-path>/MyAPI_get_by_name_like.zip",  
  "Environment": {"Variables": {"RESERVED": ""}},  
  "Permissions": [{"StatementId": "api_execute_MyAPI_GET", "Action": "lambda:InvokeFunction",  
    "Principal": "apigateway.amazonaws.com", "SourceArn": "arn:aws:execute-  
api:<region>:<account>:*/*/GET/MyPath/*"}]  
}
```



Assuming that this JSON is stored in a file named “MyAPI_last_name_like_function.json”, it can be created using:

```
python api_lambda_function_creator.py --acct <acct_num> --file <my-local-location>/MyAPI_last_name_like_function.json
```

The JSON structure for the Lambda function contains the function name, the execution role (set up previously) and assumes an S3 .zip deployment in your AWS account as specified by “DeploymentBucket” and “DeploymentKey”. “Environment” should contain any environment variables used by your Lambda (an AWS secret for Serverless Redshift connections, for example). To be executed by the AWS API Gateway, the “Permissions” entry must contain a valid source ARN, which may be set up later by creating API Gateway resources, (which may include a path as specified above). Such a path cannot be set up until the API itself exists.

API Gateway Resources

The following JSON creates a “shell” in API Gateway – a placeholder for adding methods, deployment stages, API keys, etc.:

```
{
  "APIName": "MyAPI",
  "KeySource": "HEADER",
  "EndpointType": "REGIONAL",
  "DefaultStageName": "development",
  "UsagePlanName": "MyAPI_usage_plan",
  "APIKeyName": "MyAPI_api_key"
}
```

The supplied utilities use this information to create an API Gateway REST API with no VPC. The REST API created has a REGIONAL endpoint and expects an API Key in the header. The “DefaultStageName”, “UsagePlanName” and “APIKeyName” are used when the API is finalized and deployed. A simple “shell” for the API without any resources or methods can be created with the following execution (assuming that the JSON is stored in a file named as in the example):

```
python api_gateway_shell_creator.py --file <my-local-location>/MyAPI_API.json
```

This creates a REST API with a single resource (the root path), as below:

Resources

API actions ▼

Deploy API

Create resource

/

Resource details

Update documentation

Enable CORS

Path /	Resource ID u49d265cqf
-----------	---------------------------

Methods (0)

Delete

Create method

Method type ▲	Integration type ▼	Authorization ▼	API key ▼
No methods			
No methods defined.			

A path resource is created using the identifier of the API and the identifier of the root path (in API Gateway) as illustrated below:

```
python api_gateway_path_creator.py --api-id <api-id> --parent-id <root-id> --path-text MyPath
```

If the argument provided as --parent-id is the root identifier, the AWS Console for your account should show the API as:

[API Gateway](#) > [APIs](#) > Resources - MyAPI

Resources

Create resource

☐ /
/MyPath

Before a method that accepts any parameters (for GET) or JSON (for POST, PUT, DELETE) can be created, an appropriate model must be created in AWS API Gateway so that it knows what data to expect. A simple GET that accepts a "last_name" parameter can have a simple model specification like:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "title": "MyAPIgetbynameModel",
  "properties": {
    "last_name": { "type": "string" }
  }
}
```

```
    },
    "required": ["last_name"]
}
```

The provided utility that creates models in AWS API Gateway uses “title” in the JSON structure as the name of the model. Assuming the JSON is stored in a file named “MyAPI_last_name_like_model.json”, it can be created as:

```
python api_gateway_model_creator.py --api-id <api-id> --file <my-local-
location>/MyAPI_last_name_like_model.json
```

After the model is created in API Gateway, it can be used in method creation. Methods are created by a utility using a JSON structure like:

```
{
  "MethodName": "get_by_name_like",
  "HTTPType": "GET",
  "IntegrationType": "POST",
  "URI": "arn:aws:apigateway:<region>:lambda:path/2015-03-
31/functions/arn:aws:lambda:<region>:<account>:function:MyAPI_get_by_name_like/invocations",
  "ModelName": "MyAPIgetbynameLikeModel",
  "RequestTemplate": {
    "application/json": "\"{\\\"last_name\\\": \\\"$input.params('last_name')\\\"}\""
  }
}
```

There are a few important things to note in this JSON structure. The HTTP type of the method determines the “REST” method, while the invocation type will always be “POST” for method that invoke Lambda functions. The “URI” contains the name of the method to create and ends in invocations. The “ModelName” must already exist in API Gateway and the request template indicates the expected input. Assuming that this JSON is stored in a file named “MyAPI_last_name_like_method.json”, the following execution creates the method:

```
python api_gateway_method_creator.py --acct <acct_num> --api-id <api-id> --parent-id <path-id> --
file <my-local-location>/MyAPI_last_name_like_method.json
```

If the ID provided as the parent references “MyPath”, then the AWS API Gateway console would show the API resources as:

[API Gateway](#) > [APIs](#) > Resources - MyAPI

Resources

Create resource

☐ /

☐ /MyPath

☐ /get_by_name_like

GET

Methods to implement REST POST or PUT should contain models used to describe the entire underlying object to be supplied as a body to the request. This is an example of a JSON model definition that could be used for POSTS and PUTS, assuming that the underlying data has only three attributes, two of which are required:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "title": "MyAPIFullModel",
  "properties": {
    "last_name": { "type": "string" }, "first_name": { "type": "string" }, "date_of_birth": {
"type": "string" }
  },
  "required": ["last_name", "first_name"]
}
```

This model could be used by both POST and PUT methods, which may execute different Lambda functions and may be defined in JSON as below:

```
{
  "MethodName": "add",
  "HTTPType": "POST",
  "IntegrationType": "POST",
  "URI": "arn:aws:apigateway:<region>:lambda:path/2015-03-
31/functions/arn:aws:lambda:<region>:<account>:function:MyAPI_add/invocations",
  "ModelName": "MyAPIFullModel"
}
```

If POST and PUT methods are added to the API as “add” and “modify”, it should appear in the AWS console as:

[API Gateway](#) > [APIs](#) > Resources - MyAPI

Resources

Create resource

[-] /

[-] /MyPath

[-] /add

POST

[-] /get_by_name_like

GET

[-] /modify

PUT

To finalize and deploy the API, execute the following utility with the same configuration file that was used to create the API shell:

```
python api_gateway_deployer.py --api-id <api-id> --file <my-local-location>/MyAPI_API.json
```

This command creates a deployment stage, an API key and a usage plan, associating them with each other and the API in AWS API Gateway. The API is now ready to test using curl or your tool of choice.

Summary

These utilities assist in the basic tasks required to set up a serverless API using AWS API Gateway. At a high level, there are only three simple steps:

- Create or determine IAM policies and roles to use
- Create Lambdas to access data
- Create API Gateway resources that implement the API interface by executing the Lambdas

The utilities are provided “as-is” and are intended only to provide a simple, consistent way to implement a serverless REST API via AWS API Gateway. The complexity of accessing data sources and returning correct results is assumed to be implemented via AWS Lambdas, where code for the Lambdas is complete and deployed on AWS S3 as a .zip file distribution. Code to implement a specific API is not provided with these utilities and the examples are intended to illustrate the use of the utilities, not to implement any specific API.