

# Api Gateway Implementation Report!

**Prepared by:** Falilat Owolabi

**Date:** jan 19th 2026

**Cloud provider:** AWS

**Test Environment:** AWS console and Postman

**Testing Window:** jan 14th – jan 19th 2026

**Type:** secure integration Simulation (Self-led, Part of API Security Capstone)

## Format

This report shows implementation of an API gateway using AWS cloud service provider which integrates some security mechanisms to protect the API endpoint use in testing which include:

- Https integration
- Secured with jwt token

## 1. Executive Summary

This report documents the steps taken to implement AWS APIgateway with a simple API endpoint, it shows how some security configuration can be implemented to mitigate against security vulnerability and protect the API endpoint through the API gateway.

Additionally, through frontend JWT Authorization and TLS integrated, a protected API gateway was achieved.

## Methodology

The testing approach solely use the AWS console for setting up and implementation and POSTMAN was used for testing the result of the configuration

## Lambda Function

An API gateway with an API endpoint on AWS involves having a function/logic you can test your implementation with and AWS has a console that allows us to easily create a Lambda function that we can integrate into our API called Lambda function. After going through the necessary steps in AWS console to create a lambda function, I was able to create a simple function that can be used as shown below:

The screenshot shows the AWS Lambda console interface. At the top, there's a navigation bar with tabs for 'Code', 'Test', 'Monitor', 'Configuration', 'Aliases', and 'Versions'. Below this, the 'Function overview' section displays the function name 'Api-Security-Training' and a 'Layers' section indicating '(0)'. On the right side, there are buttons for 'Throttle', 'Copy ARN', 'Actions', 'Export to Infrastructure Composer', and 'Download'. The 'Description' field is empty, and the 'Last modified' field shows '16 seconds ago'. The 'Function ARN' is listed as 'arn:aws:lambda:eu-north-1:361966321984:function:Api-Security-Training'. The 'Function URL' is also listed. Below these details, there's a code editor window titled 'index.mjs' containing the following JavaScript code:

```

JS index.mjs x
index.mjs > handler > response > body
1 export const handler = async (event) => {
2   // TODO implement
3   const response = {
4     statusCode: 200,
5     body: JSON.stringify('Hello Lambda, from API security'),
6   };
7   return response;
8 };
9

```

The code editor includes standard file operations like 'Open in Visual Studio Code' and 'Upload from'. On the left, there's an 'EXPLORER' sidebar showing the file 'index.mjs' under 'API-SECURITY-TRAINING'. Below it, there are sections for 'DEPLOY' (with 'Deploy' and 'Test' buttons) and 'TEST EVENTS' (with a '+ Create new test event' button). A message at the bottom right says 'Successfully updated the function Api-Security-Training.'

Lambda console after creating lambda function.

Testing the lambda function to be sure

**Executing function: succeeded (Logs [L](#))**

**Details**

```
{
  "statusCode": 200,
  "body": "\Hello Lambda, from API security\"
}
```

**Summary**

Code SHA-256 b3QWX8x/00A4HridVuuueY7EqOUAvTtgddGR55bMx2w=	Execution time 1 second ago
Function version \$LATEST	Request ID c131bfc1-7f7d-45b3-af2a-f665ebb3fbaf
Duration 8.46 ms	Billed duration 151 ms
Resources configured 128 MB	Max memory used 79 MB
Init duration 142.30 ms	

**Log output**

The area below shows the last 4 KB of the execution log. [Click here L](#) to view the corresponding CloudWatch log group.

```
START RequestId: c131bfc1-7f7d-45b3-af2a-f665ebb3fbaf Version: $LATEST
END RequestId: c131bfc1-7f7d-45b3-af2a-f665ebb3fbaf
REPORT RequestId: c131bfc1-7f7d-45b3-af2a-f665ebb3fbaf Duration: 8.46 ms Billed Duration: 151 ms Memory Size: 128 MB Max Memory Used: 79 MB Init Duration: 142.30 ms
```

**Test event info**

Invoke your function without saving an event, configure the JSON event, then choose Test.

**Test event action**

Create new event       Edit saved event

**Invocation type**

Synchronous  
Executes the Lambda function and blocks until receiving the function's response, with a maximum timeout of 15 minutes. Returns function output or error details directly to the calling application.

Asynchronous  
Enqueues the Lambda function for execution and returns immediately with a request ID. Function processes independently, with results optionally sent to a configured destination like SQS, SNS, or EventBridge.

**Event name**

Maximum of 25 characters consisting of letters, numbers, dots, hyphens and underscores.

**Event sharing settings**

Private  
This event is only available in the Lambda console and to the event creator. You can configure a total of 10. [Learn more L](#)

Shareable  
This event is available to IAM users within the same account who have permissions to access and use shareable events. [Learn more L](#)

[CloudWatch Logs Live Tail](#)   [Save](#)   [Test](#)

## Build with Rest API

The next step is to build a rest Api, that we will use with our lambda function, the flow below shows

**Create REST API** Info

**API details**

- New API Create a new REST API.
- Clone existing API Create a copy of an API in this AWS account.
- Import API Import an API from an OpenAPI definition.
- Example API Learn about API Gateway with an example API.

**API name**  
Rest-API

**Description - optional**  
testing for API Security

**API endpoint type**  
Regional APIs are deployed in the current AWS Region. Edge-optimized APIs route requests to the nearest CloudFront Point of Presence. Private APIs are only accessible from VPCs.

**Regional**

**Security policy - new** | Info  
Transport Layer Security (TLS) protects data in transit between a client and server. The security policy also determines the cipher suite options that clients can use with your API.

**SecurityPolicy\_TLS13\_1\_2\_2021\_06** 

**Endpoint access mode - new** | Info  
Provide additional governance for your APIs.

Basic Allow all clients to access the API

Strict (recommended) Enforce Server Name Indication (SNI) validation

**IP address type** | Info  
Select the type of IP addresses that can invoke the default endpoint for your API.

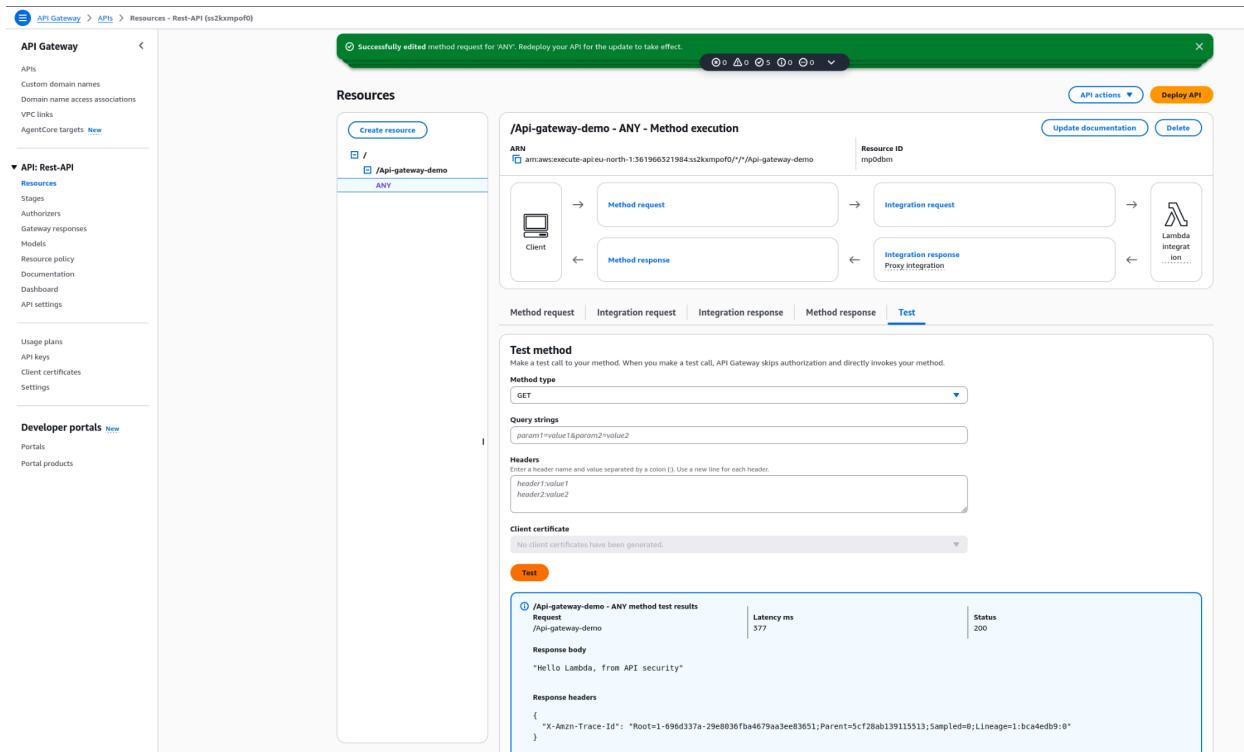
IPv4 Supports only edge-optimized and Regional API endpoint types.

Dualstack Supports all API endpoint types.

**Create API**

Creating restAPI flow with the integration of TLS, to make sure communication between the client and gateway is secured.

After that, I created a resource and also create a method that will be used for the api, where I am able to integrate the lamda function created above, tested that it is working fine and afterwards, I have the Output below ready to be deployed



The deployment is done to the prod stage as seen above.

## Testing The the invoke url in post man

Now that the API gateway has been deployed, I then copied the invoke url and past in postman to confirm it can be access properly

The screenshot shows a POSTMAN interface with the following details:

- Method:** GET
- URL:** https://itxec402r8.execute-api.eu-north-1.amazonaws.com/prod (highlighted with a red arrow)
- Headers:** (6) (highlighted with a red arrow)
- Body:** (JSON) { } (highlighted with a red arrow)
- Response:**
  - Status: 200 OK
  - Time: 1.08 s
  - Size: 369 B
  - Content: "Hello Lambda, from API security" (highlighted with a red arrow)

## Using JWT as the AuthorizationToken to Access the endpoint

Now that the API gateway is working well, the next phase is to add an Authorizer that can help secure it from being accessed by unauthorised user which will lead us to having to create an authorizer lambda function that will be added to our Rest API

**Create function** Info

Choose one of the following options to create your function.

**Author from scratch**  
Start with a simple Hello World example.

**Use a blueprint**  
Build a Lambda application from sample code and configuration presets for common use cases.

**Container image**  
Select a container image to deploy for your function.

**Basic information**

**Function name**  
Enter a name that describes the purpose of your function.

**Runtime** Info  
Choose the language to use to write your function. Note that the console code editor supports only Node.js, Python, and Ruby.  
 ▼ (C) Last fetched 19/01/2026, 07:26:53

**Durable execution - new** Info  
Enable durable execution to simplify building resilient multi-step applications that checkpoint progress and resume after interruptions. Supports Python and Node.js runtimes. [View pricing](#) ↗  
 Enable

**Architecture** Info  
Choose the instruction set architecture you want for your function code.  
 arm64  
 x86\_64

**Permissions** Info  
By default, Lambda will create an execution role with permissions to upload logs to Amazon CloudWatch Logs. You can customize this default role later when adding triggers.

**Change default execution role**

**Additional configurations**  
Use additional configurations to set up networking, security, and governance for your function. These settings help secure and customize your Lambda function deployment.

Cancel “ Create function

Creating the authorization lambda function

```
js index.mjs > ⚡ handler > ⚡ handler > [!] token
1  exports.handler = async (event) => {
2    const token = event.authorizationToken;      Amazon Q Tip 1/3: Start typing to get
3    // In a real scenario, validate the token (e.g., JWT verification, database lookup)
4    if (token === "secretToken") {
5      return generatePolicy('user', 'Allow', event.methodArn);
6    } else {
7      return generatePolicy('user', 'Deny', event.methodArn);
8    }
9  };
10
11 const generatePolicy = (principalId, effect, resource) => {
12   return {
13     principalId,
14     policyDocument: {
15       Version: '2012-10-17',
16       Statement: [
17         {
18           Action: 'execute-api:Invoke',
19           Effect: effect,
20           Resource: resource
21         }
22       ],
23       // Optional: Pass data to the backend Lambda function via 'context'
24       // context: {
25       //   "userId": "123"
26     }
27   };
28 }
```

I got this code online when I searched, edit it to suit my purpose and here I realised I need a token and I use [jwt.io](https://jwt.io) to generate a new token

The screenshot shows the jwt.io interface. On the left, under 'Encoded Value', a JWT token is displayed: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IiwiZW5jb2RlcyI6IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKf2QT4fwMeJf36P0koyJV\_adQssw5c. The token is highlighted with a green border. Below it, a green button says 'Valid Signed JWT'. On the right, there are three sections: 'Decoded Header' (JSON tab selected), 'Decoded Payload' (JSON tab selected), and 'JWT Signature Verification' (optional). The 'Decoded Header' shows { "alg": "HS256", "typ": "JWT" }. The 'Decoded Payload' shows { "sub": "1234567890", "name": "John Doe", "iat": "1516239022" }. Both sections have a 'Valid JSON' button.

I copied the token from here and past it in my lambda function, deployed it and tested it to confirm it is working as expected

The screenshot shows the AWS Lambda Authorizer configuration and a simulated invocation test.

**Authorizer ID:** vmlzqx

**Lambda function:** API-Authorizer (eu-north-1)

**Lambda invoke role - optional:** -

**Lambda event payload:** Token

**Token source:** authorizationToken

**Token validation - optional:** None

**Authorization caching:** 300 seconds

**Test authorizer:** Test your authorizer with a simulated invocation request.

**Token source:** authorizationToken

**Token value:** eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IiwiZW5jb2RlcyI6IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKf2QT4fwMeJf36P0koyJV\_adQssw5c

**Authorizer test: jwt-Auth**

```

200
Policy
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": "execute-api:Invoke",
      "Effect": "Allow",
      "Resource": "arn:aws:execute-api:eu-north-1:361966321984:itxec402r8/ESTestInvoke-stage/GET/"
    }
  ]
}

Log
Execution log for request f369faca-d9d3-47c8-bd04-e23d78aeb9e8 Mon Jan 19 07:12:04 UTC 2026 : Starting authorizer: vmlzqx for request: f369faca-d9d3-47c8-bd04-e23d78aeb9e8 Mon Jan 19 07:12:04 UTC 2026 : Incoming identity: ****Qssw5c Mon Jan 19 07:12:04 UTC 2026 : Endpoint request URI: https://lambda.eu-north-1.amazonaws.com/2015-03-31/functions/arn:aws:lambda:eu-north-1:361966321984:function:API-Authorizer/invocations Mon Jan 19 07:12:04 UTC 2026 : Endpoint request headers: X-Amz-Date=20260119T071204Z, X-amzn-apigateway-api-id=itxec402r8, Accept=application/json, User-Agent=AmazonAPIGateway_itxec402r8, Host=lambda.eu-north-1.amazonaws.com, X-Amz-Content-Sha256=f9734049277cc24cf5699a69b01e2c6cf0e4e7e27e45abb118dfbe0d10b21a, X-Amzn-Trace-Id=Root=1-696dd944-90ab965af6735e70f1ac4788, x-amzn-lambda-integration-tag=f369faca-d9d3-47c8-bd04-e23d78aeb9e8, Authorization=****a9b2a5, X-Amz-Source-Arn=arn:aws:execute-api:eu-north-1:361966321984:itxec402r8/authorizers/vmlzqx, X-Amz-Security-Token=IQoJb3Jp2luX2jJEMX//////////wEaCmV1LW5vncnRoTEISD8GAiAm2da9fGeu7b9jLkOVAvk9K022Pazzp98Kkx-6lIClC0Cpfv99/rBEx+PG5Yp7uH2357JPJu12reIS5Q50UHyZy [TRUNCATED] Mon Jan 19 07:12:04 UTC 2026 : Endpoint request body after transformations: {"type": "TOKEN", "methodArn": "arn:aws:execute-api:eu-north-1:361966321984:itxec402r8/ESTestInvoke-stage/GET/", "authorizationToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IiwiZW5jb2RlcyI6IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKf2QT4fwMeJf36P0koyJV_adQssw5c"} Mon Jan 19 07:12:04 UTC 2026 : Sending request to https://lambda.eu-north-1.amazonaws.com/2015-03-31/functions/arn:aws:lambda:eu-north-1:361966321984:function:API-Authorizer/invocations Mon Jan 19 07:12:04 UTC 2026 : Authorizer result body before parsing: {"principalId": "test", "policyDocument": {"Version": "2012-10-17", "Statement": [{"Action": "execute-api:Invoke", "Effect": "Allow", "Resource": "arn:aws:execute-api:eu-north-1:361966321984:itxec402r8/ESTestInvoke-stage/GET/"}]} Mon Jan 19 07:12:04 UTC 2026 : Using valid authorizer policy for principal: ****user

```

Back to the API gateway console, in my rest API created earlier, I clicked on

Authorizer to add the lambda function created for authorizer, to the ApiGateway as seen below

Create authorizer [Info](#)

**Authorizer details**

Authorizer name

Authorizer type [Info](#)  
Choose to authorize your API calls using one of your Lambda functions or a Cognito User Pool.  
 Lambda  
 Cognito

Lambda function  
Provide the Lambda function name or alias. You can also provide an ARN from another account.  
  [X](#)

Grant API Gateway permission to invoke your Lambda function  
When you save your changes, API Gateway updates your Lambda function's resource-based policy to allow this API to invoke it.

Lambda invoke role - *optional*  
Specify an optional role API Gateway will use to make requests to your authorizer. For optimal API performance it is strongly recommended to activate Regional STS in the region where your API is located.

Lambda event payload  
Choose token to send a single header that contains an authorization token. Choose request to send all request parameters.  
 Token  
 Request

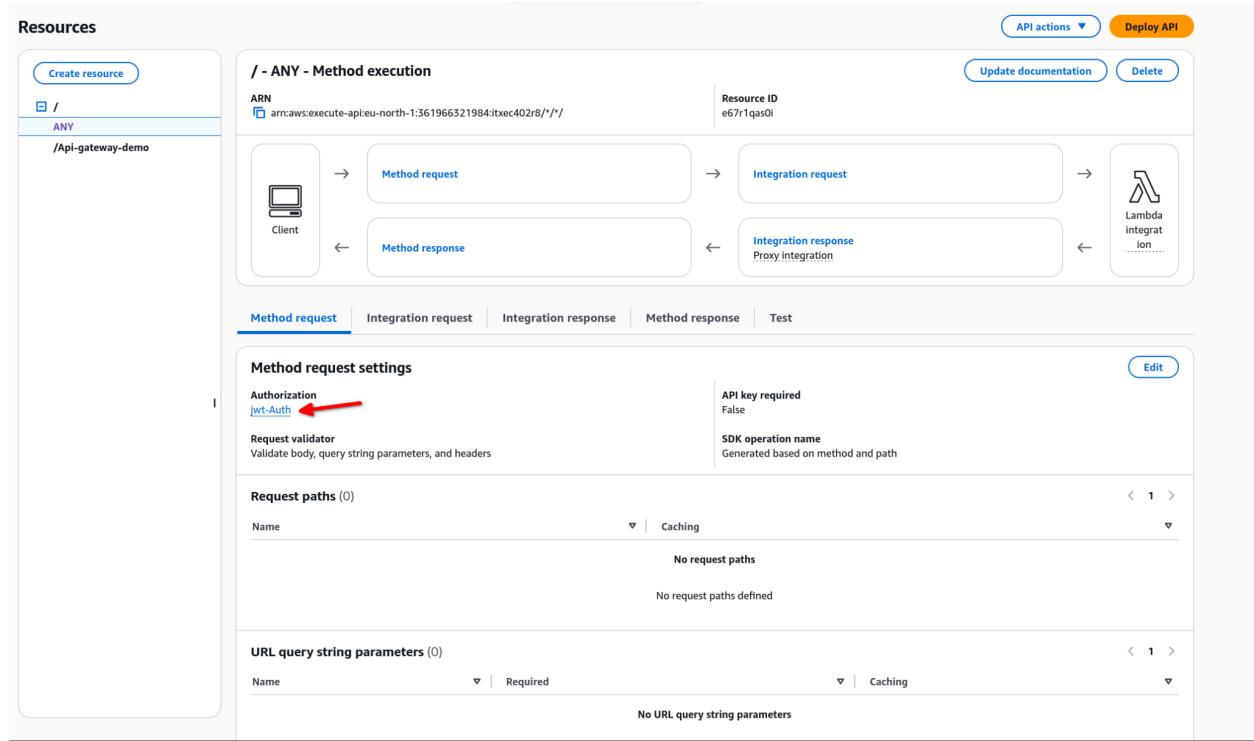
Token source  
Enter the header that contains the authorization token.

Token validation - *optional*  
Enter a regular expression to validate tokens.

TTL (1-3600 seconds)  
 [?](#)

[Cancel](#) [Create authorizer](#)

After the creation of Authorization in the API gateway, I then go back to the AP-gateway-demo and I changed the Authorization from Non to the JWT-Auth I created and redeployed the API gateway as shown below



After this I then copied the invoke url and paste in postman for testing to see if it will be denied without the right Authorization token and allowed with the right Authorization token as seen below respectively.

The screenshot shows the Postman application interface. At the top, it displays the URL `https://itxec402r8.execute-api.eu-north-1.amazonaws.com/prod`. The method is set to `PATCH`, and the target URL is also `https://itxec402r8.execute-api.eu-north-1.amazonaws.com/prod`. On the right side, there are buttons for `Save`, `Share`, and a blue `Send` button.

The main workspace has tabs for `Docs`, `Params`, `Authorization`, `Headers (7)`, `Body` (which is currently selected), `Scripts`, and `Settings`. Below these tabs, there are radio buttons for `none`, `form-data`, `x-www-form-urlencoded`, `raw` (which is selected), `binary`, `GraphQL`, and `JSON`.

The `Body` section contains the following JSON payload:

```
1 Ctrl+Alt+P to Ask AI
```

At the bottom, there are tabs for `Body`, `Cookies`, `Headers (7)`, `Test Results`, and a refresh icon. The status bar at the bottom right shows a `401 Unauthorized` error, a response time of `683 ms`, and a size of `299 B`.

The screenshot shows the Postman application interface. At the top, there are tabs for 'Docs', 'Params', 'Authorization', 'Headers (8)', 'Body', 'Scripts', and 'Settings'. The 'Headers' tab is selected, displaying a list of header fields with their values. Below the headers, there are tabs for 'Body', 'Cookies', 'Headers (7)', and 'Test Results'. The 'Body' tab is selected, showing a JSON response: { "Hello Lambda, from API security" }. At the bottom right, the status bar indicates '200 OK', '1.16 s', '369 B', and other metrics.

## Conclusion

The Implementation of API gateway introduced me to the cloud space and how services are rolling out features to save APIs and infrastructure . The protection each service has put in place and from a security perspective, what could go wrong if the protection is not in place in the API, API gateway in my opinion added another layer of protection to data in the cloud, protecting malicious injection in the system.

## Challenges faced:

I struggled, understanding how to go about it. After so much research I was able to come up with this implementation even with the fact that I could not implement all features expected to be implemented but in the end I have a better understanding of the cloud service and what my role will be as an API tester.