

Deploying an Application in a Private Subnet EC2, Attaching an Internal NLB, API Gateway through VPC link and Configuring Custom Domain Name For end user

The AWS services we require to perform the task are

1. VPC
 - 1.1. Subnet
 - 1.2. Route Table
 - 1.3. Internet Gateway
 - 1.4. NAT Gateway
 - 1.5. EC2
2. Network Load Balancer (NLB)
 - 2.1. Target Groups routing
 - 2.2. VPC & Subnet selection
 - 2.3. Internal / Internet Facing
3. API Gateway
 - 3.1. Type (REST/HTTP/Web-Socket)
 - 3.1.1. Resources
 - 3.1.2. Methods
 - 3.1.3. Integrations
 - 3.1.4. Stages
 - 3.1.5. Authorizers (IAM, Lambda, Cognito)
 - 3.1.6. API keys
 - 3.1.7. Logs
 - 3.1.8. Responses
 - 3.2. VPC Links
 - 3.3. Custom Domain Names
 - 3.3.1. ACM
 - 3.3.2. Route 53
 - 3.3.2.1. Hosted Zone
 - 3.3.2.2. Name Servers
 - 3.3.2.3. DNS Record

VPC- Subnet configuration:

- Create a VPC and 2 Subnets inside that VPC.
- Public Subnet will have a Public IP and can Communicate with internet via Internet Gateway
- Private Subnet won't have a Public IP. We need to assign a NAT gateway in public subnet and instance in private subnet can communicate to internet via NAT gateway.

Our application is deployed in the ec2 of private subnet.

Target Group:

load balancer routes requests to the targets in a target group and performs health checks on the targets.

- Create a Target Group and choose EC2 as target type.
- Select your VPC and HTTP protocol. Click next.
- Register targets by selecting targets from Available instances section. Click on create target group.

NLB configuration:

- In this project we are going to configure an Internal NLB. An internal load balancer routes requests from clients to targets using private IP addresses.

Basic configuration

Load balancer name
Name must be unique within your AWS account and cannot be changed after the load balancer is created.

InternalLB

A maximum of 32 alphanumeric characters including hyphens are allowed, but the name must not begin or end with a hyphen.

Scheme
Scheme cannot be changed after the load balancer is created.

☐ Internet-facing
An internet-facing load balancer routes requests from clients over the internet to targets. Requires a public subnet. [Learn more](#)

☒ Internal
An internal load balancer routes requests from clients to targets using private IP addresses.

IP address type [Info](#)
Select the type of IP addresses that your subnets use.

☒ IPv4
Recommended for internal load balancers.

☐ Dualstack
Includes IPv4 and IPv6 addresses.

- Network Mapping. Select the VPC and private subnet.

Network mapping [Info](#)

The load balancer routes traffic to targets in the selected subnets, and in accordance with your IP address settings.

VPC
Select the virtual private cloud (VPC) for your targets. The selected VPC cannot be changed after the load balancer is created. To confirm the VPC for your targets, view your [target groups](#).

Sample-VPC
vpc-06f88a3cca2d9ae9d
IPv4: 10.0.0.0/16

Mappings
Select at least one Availability Zone and one subnet for each zone. We recommend selecting at least two Availability Zones. The load balancer will route traffic only to targets in the selected Availability Zones. Zones that are not supported by the load balancer or VPC cannot be selected. Subnets can be added, but not removed, once a load balancer is created.

☒ us-east-1c (use1-az2)

Subnet
subnet-078444258aca944d8 Private

IPv4 settings

IPv4 address
Assigned from CIDR 10.0.1.0/24

Private IPv4 address
Assigned from CIDR 10.0.1.0/24

- In listener and routing select the Target Group. The rules that you define for a listener determine how the load balancer routes requests to its registered targets.

Listeners and routing [Info](#)

A listener is a process that checks for connection requests using the port and protocol you configure. The rules that you define for a listener determine how the load balancer routes requests to its registered targets.

▼ Listener TCP:80 [Remove](#)

Protocol: TCP Port: 80 Default action: Forward to Select a target group [Create target group](#)

1-65535

Listener tags - optional
Consider adding tags to your listener. Tags enable you to categorize your AWS resources so you can more easily manage them.

[Add listener tag](#)
You can add up to 50 more tags.

[Add listener](#)

- Create an Internal NLB. Keep a note of DNS name of load balancer. It will be used in VPC links of API Gateway.

InternalLoadBalancer [Actions](#)

▼ **Details**

arn:aws:elasticloadbalancing:us-east-1:935342901096:loadbalancer/net/internal-loadbalancer/03009fc8356e71

Load balancer type Network	DNS name InternalLoadBalancer-cb3009fc8356e71.elb.us-east-1.amazonaws.com (A Record)	Status Active	VPC vpc-06f88a3cca2d9ae9d
IP address type IPv4	Scheme Internal	Availability Zones subnet-078444258aca944d8 us-east-1c (use1-az2)	Hosted zone Z26RNL4HYFTOT1
Date created February 17, 2023, 11:30 (UTC+05:30)			

[Listeners](#) [Network mapping](#) [Monitoring](#) [Integrations](#) [Attributes](#) [Tags](#)

Listeners (1) [Actions](#) [Add listener](#)

A listener checks for connection requests on its port and protocol. Traffic received by the listener is routed according to its rules.

Search

Protocol/Port	ARN	Security policy	Default SSL cert	Default routing rule	ALPN policy	Tags
TCP:80	arn:aws:elasticloadbalancing:us-east-1:935342901096:loadbalancer/net/internal-loadbalancer/03009fc8356e71	Not applicable	Not applicable	Forward to SampleTargetGroup	None	0

API Gateway Configuration:

- Here we are configuring both HTTP and REST Api.
- For HTTP API Create a VPC link.

The screenshot shows the 'Create a VPC link' configuration page in the AWS API Gateway console. The 'VPC Link details' section includes a 'Name' field and a 'VPC' dropdown menu. The 'Subnets' section displays a table of available subnets. The 'Security groups' section displays a table of available security groups. The 'Tags' section is at the bottom.

Subnet	Name	Availability Zone	Subnet ID
subnet-01234567890123456	Public	us-east-1c	subnet-01234567
subnet-01234567890123456	Private	us-east-1c	subnet-01234567

Security group	Name	Rules
sg-01234567890123456	sg-01234567890123456	Allow HTTP traffic from the internet
sg-01234567890123456	sg-01234567890123456	Allow HTTP traffic from the internet
sg-01234567890123456	sg-01234567890123456	Allow HTTP traffic from the internet

- Now build a HTTP Api and Specify the backend services that your API will communicate with. These are called integrations. For HTTP integration, API Gateway sends the request to the URL that you specify and returns the response from the URL.
- Configure routes, Define Stages and create an API gateway.
- Similarly for REST API. Create an API with defining Resources, Methods, Integrations. Deploy it to the stage.
- Now we will get Invoke URL for the API gateway. This is the default end point where end users can access our API. We can Setup Custom Domain Names and disable default end point.

Custom Domain Names:

To setup of Custom Domain name, we need to have

- Route 53 Nameserver for DNS.
- Wild card certificates should be stored in ACM. Else we can request ACM for a certificate with DNS verification.

Now Create a Custom Domain Name and provide certificates.

The screenshot shows the AWS API Gateway console. The 'Domain details' section includes a 'Domain name' field with the value 'test.subhenduadivthcloudlearn.xyz', a 'Minimum TLS version' section with radio buttons for 'TLS 1.2 (recommended)', 'TLS 1.0 (supports only REST APIs)', and 'Mutual TLS authentication', and a 'Mutual TLS authentication' section with a checkbox and a link to 'Learn more'. The 'Endpoint configuration' section includes an 'Endpoint type' section with radio buttons for 'Regional' (selected) and 'Edge-optimized (supports only REST APIs)', and an 'ACM certificate' section with a dropdown menu showing 'test.subhenduadivthcloudlearn.xyz' and a 'Create a new ACM certificate' link.

- Configure Path for the CDN and update Route53 DNS A records(alias) to API Gateway.

We can access our Api from Custom URL from a Browser or Postman.

Attaching Authorizers to Api Gateway:

We can authorize gateway in 3 methods.

- API Keys
- Lambda Authorizer
- Cognito Authorizer

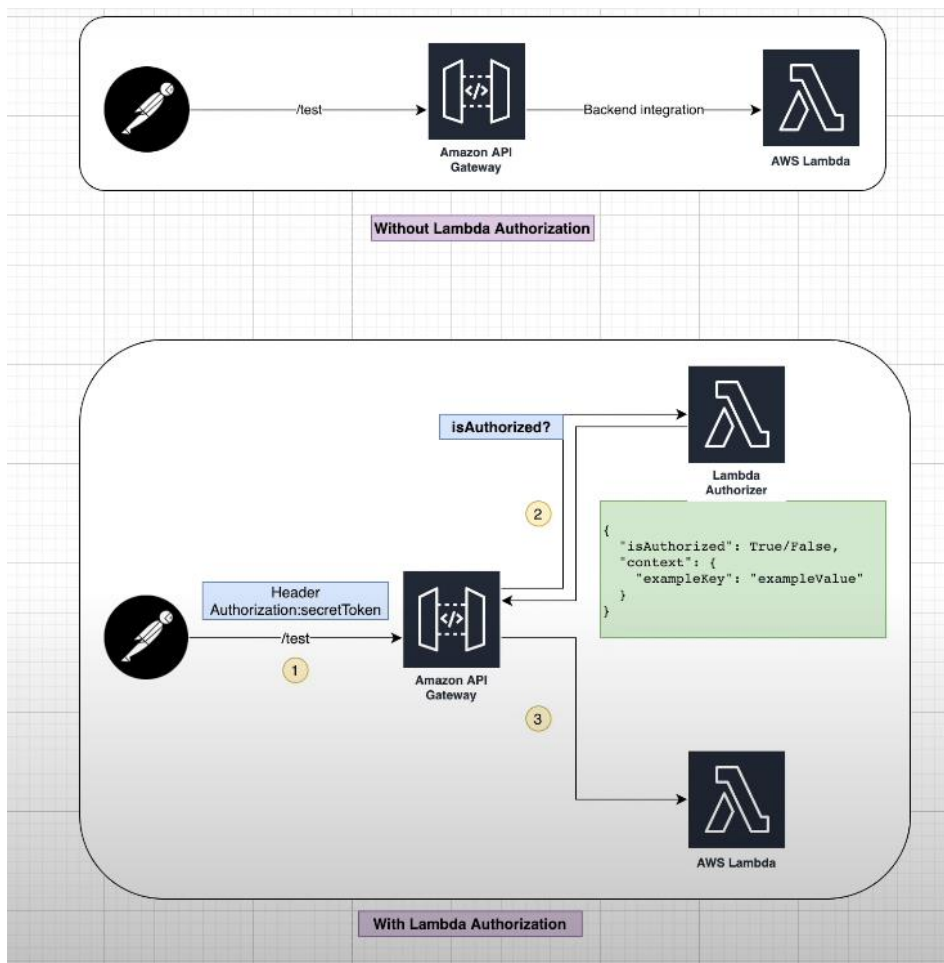
API Keys:

- Create Usage Plan. Usage plan helps us in metering our API, we can enable throttling and quota limit on each API key.

The screenshot shows the 'Create Usage Plan' form in the AWS API Gateway console. The form has a 'Name' field with the value 'basic' and a 'Description' field. The 'Throttling' section has a checkbox for 'Enable throttling' which is checked, and fields for 'Rate' (requests per second) and 'Burst' (requests). The 'Quota' section has a checkbox for 'Enable quota' which is checked, and a field for 'requests per' with a dropdown for 'Month'. A 'Next' button is at the bottom right.

- Associate API stages to the Usage plan, Subscribers will only be allowed to access the API stages associated to their usage plan.
- In next step we have to subscribe an API key to the usage plan. The API key will be provided to our customers.
- We can create an API Key and add it to usage plans.
- Go to API resources, select method, in method request section set API Key required to true.
- Next step, Deploy the API, go to API keys section and copy the API key, test it with postman.

Lambda Authorizer:



- The client calls a method on an API Gateway API method, passing a bearer token or request parameters.
- API Gateway checks whether a Lambda authorizer is configured for the method. If it is, API Gateway calls the Lambda function.
- If the call succeeds, the Lambda function grants access by returning an output object containing at least an IAM policy and a principal identifier.

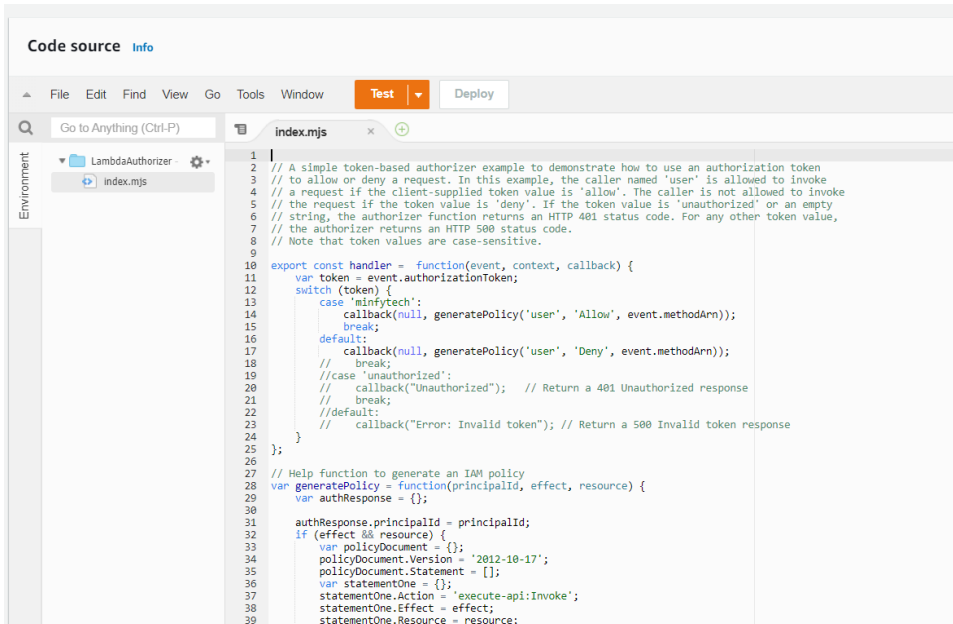
- API Gateway evaluates the policy.
 - If access is denied, API Gateway returns a suitable HTTP status code, such as 403 ACCESS_DENIED.
 - If access is allowed, API Gateway executes the method. If caching is enabled in the authorizer settings, API Gateway also caches the policy so that the Lambda authorizer function doesn't need to be invoked again.

Configuring Lambda Function as an Authorizer:

Let's create a token-based lambda authorizer function.

- In the Lambda console, choose **Create function**.
- Choose **Author from scratch**.
- Choose **Author from scratch**.
- Choose **Create function**.

The following code Helps us to verify the request, If the token passed in authorized or not.



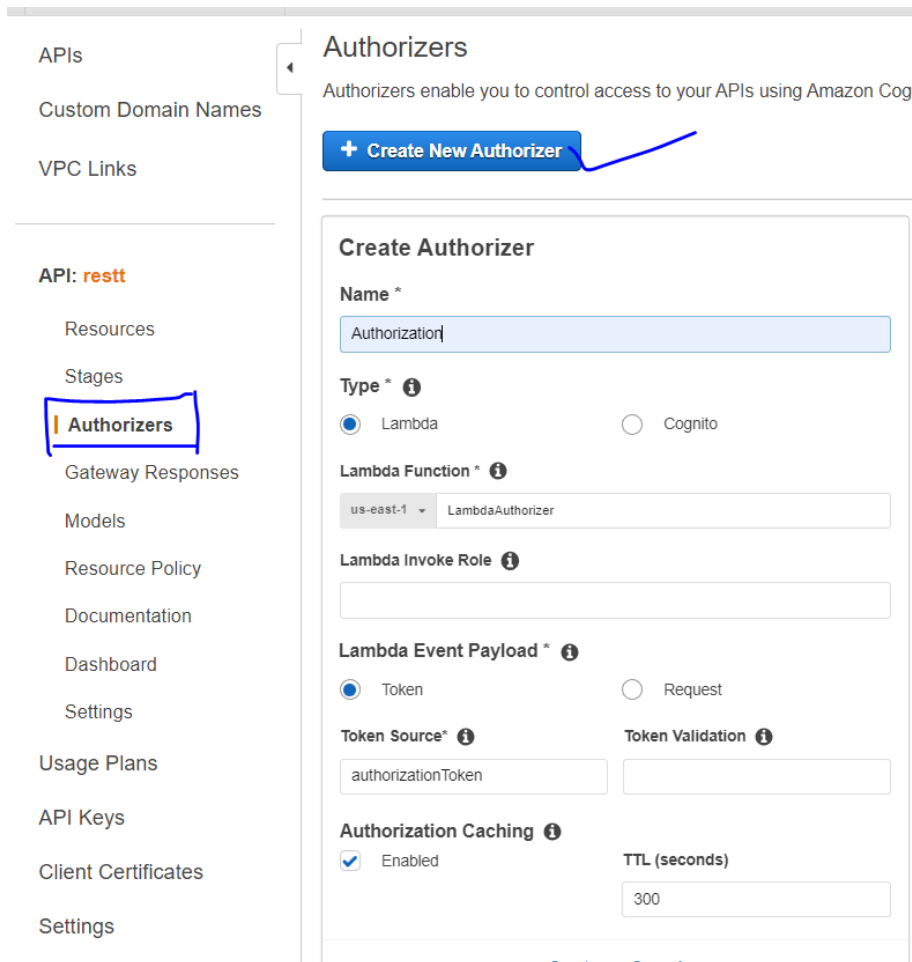
The screenshot shows a code editor with a file named 'index.mjs'. The code is a JavaScript function that acts as a token-based authorizer. It takes an event, context, and callback as arguments. It checks the 'authorizationToken' from the event. If the token is 'minifytech', it calls 'generatePolicy' with 'user' and 'Allow'. If the token is 'deny', it calls 'generatePolicy' with 'user' and 'Deny'. For any other token, it returns a 401 Unauthorized response. The 'generatePolicy' function is a helper that creates an IAM policy document with a version of '2012-10-17', a statement to 'execute-api:Invoke', and the effect set to the provided 'effect'.

```

1 // A simple token-based authorizer example to demonstrate how to use an authorization token
2 // to allow or deny a request. In this example, the caller named 'user' is allowed to invoke
3 // a request if the client-supplied token value is 'allow'. The caller is not allowed to invoke
4 // the request if the token value is 'deny'. If the token value is 'unauthorized' or an empty
5 // string, the authorizer function returns an HTTP 401 status code. For any other token value,
6 // the authorizer returns an HTTP 500 status code.
7 // Note that token values are case-sensitive.
8
9
10 export const handler = function(event, context, callback) {
11   var token = event.authorizationToken;
12   switch (token) {
13     case 'minifytech':
14       callback(null, generatePolicy('user', 'Allow', event.methodArn));
15       break;
16     default:
17       callback(null, generatePolicy('user', 'Deny', event.methodArn));
18       // break;
19       //case 'unauthorized':
20       //  callback("Unauthorized"); // Return a 401 Unauthorized response
21       //  break;
22       //default:
23       //  callback("Error: Invalid token"); // Return a 500 Invalid token response
24   }
25 }
26
27 // Help function to generate an IAM policy
28 var generatePolicy = function(principalId, effect, resource) {
29   var authResponse = {};
30
31   authResponse.principalId = principalId;
32   if (effect && resource) {
33     var policyDocument = {};
34     policyDocument.Version = '2012-10-17';
35     policyDocument.Statement = [];
36     var statementOne = {};
37     statementOne.Action = 'execute-api:Invoke';
38     statementOne.Effect = effect;
39     statementOne.Resource = resource;

```

Now move to API Gateway and select our API, Navigate to Authorizers, Create a New Authorizer.



APIs

Custom Domain Names

VPC Links

API: **restt**

Resources

Stages

Authorizers

Gateway Responses

Models

Resource Policy

Documentation

Dashboard

Settings

Usage Plans

API Keys

Client Certificates

Settings

Authorizers

Authorizers enable you to control access to your APIs using Amazon Cog

[+ Create New Authorizer](#)

Create Authorizer

Name *

Authorization

Type * ⓘ

☒ Lambda ☐ Cognito

Lambda Function * ⓘ

us-east-1 LambdaAuthorizer

Lambda Invoke Role ⓘ

Lambda Event Payload * ⓘ

☒ Token ☐ Request

Token Source* ⓘ **Token Validation** ⓘ

authorizationToken

Authorization Caching ⓘ

☒ Enabled **TTL (seconds)**

300

- Now Navigate to resources -> methods -> Method Request, Select Lambda as authorizer and save.
- Next deploy the API to effect the changes.
We can check the same via Postman.

Cognito Authorizer:

- Your app users can sign in to your user pool with a user name and password, or sign in with a third-party identity provider.
- Once the user signs-in through the sign in console provided by Cognito, it returns an Authorization token (JWT), using this token a user can access our backend resources.

First let us create a Cognito user pool,

- Configure sign-in and sign-up experience.
- Configure Security requirements.
- Integrate app with a callback URL.

- Now let's create a user from our end (user can also self-register themselves from the Hosted UI).
 - Once user is created and verified sign-up is done, next we need sign in.
 - Once a user signs in into the Cognito user pool, user can obtain a Token, using this token a user can access our backend resources.
-
- Now move to API Gateway and select our API, Navigate to Authorizers, Create a New Authorizer.
 - Now Navigate to resources -> methods -> Method Request, Select Cognito as authorizer and save.
 - Deploy API

We can test the same from postman.

Subhendu Sekhar Patro

Devops Engineer- MInfy Tech