



Perform request payload validation of REST API



Payload Validation of REST API

👤 Created by	👤 Niyush Bjr
🕒 Created time	@22 December 2025 22:57
⭐ level	Intermediate
📝 Description	Payload Validation of REST API with CRUD functionality.
☁ AWS Services	API Gateway DynamoDB Lambda
🌐 Other tools	Postman
🏷 Category	Application-Integration
☕ Buy-Me-a-Coffee	http://buymeacoffee.com/niyushbjr1L

■ □ □ * ❁.田 ★ ● ⇒ □

Perform request payload validation of REST API.

i **API Gateway** : It allows developers to expose REST API endpoint and in turns allows to integrate with multiple backend systems such as Lambda Function.
It has a feature to enable the validation — before it forwards the request to backend system.

Perform request payload validation of REST API.

What this project aims?

Key Learning Objectives :

⌘ Step 1 : Creating DynamoDB table.

⌘ Step 2 : Create Lambda Function

⌘ Step 2a : Adding DynamoDB permissions to Lambda Execution role.

⌘ Step 2b : Add the lambda code.

⌘ Step 3 : Setting up the API GateWay.

⌘ Step 4 : Create a Model.

⌘ Step 5 : Create a Resource & Enable Model for the API.

⌘ Step 5a : Set Up Request Validator

⌘ Step 5b : Apply Validator to POST Method

⌘ Step 6 : Deploy the API.

⌘ Step 7 : Test and Validate the API Endpoint with Request Payload.

Test #1 : No Request Body payload.

Expected Response.

Test #2 : Wrong Schema Parameters.

Expected Response.

Test #3 : Correct Schema Parameters.

Expected Response.

Test #4: Missing Required Field

Expected Response :

Test #5 : Wrong Data Type.

Expected Response :

⌘ Troubleshooting

Issue: Getting 403 Forbidden

Issue: Getting 404 Not Found

Issue: Validation not working (accepting invalid data)

⌘ Step 8 : Deletion Time.

Delete the API Gateway

Delete the Model (Optional)

Delete the DynamoDB

Delete the Lambda Function and Role associated with it.

Delete the Role associated with it Lambda Function

Delete the Role associated with it Lambda Function

⌘ Key Takeaways

▼ □ What this project aims?

This project demonstrates how to enable request payload validation in API Gateway before requests are forwarded to backend systems. I focuses on exposing REST API endpoints through API Gateway that integrates with Lambda functions.

▼ □ Key Learning Objectives :



1. Request Validation Fundamentals:

- Understanding how to validate that request payloads contain all required fields, and that fields follow expected patterns.
- Learning to specify validation rules in a request validator and assign the validator to individual API methods.

2. Technical Implementation :

- Creating JSON Schema models in API Gateway to define payload structure.
- Validating request bodies by specifying a model schema and verifying that the required parameters are valid and not null.

❑ Importance of payload validation :

- **Cost optimization** :
 - It's better to keep business logic separate from validation checks, and requests can be rejected at the API Gateway level if payloads are invalid, saving Lambda execution costs.
- **Code Quality & Maintainability.**
 - Implementing basic validation of input data is one of the necessary but least exciting aspects of building a robust REST API
 - Reduces code bloat by eliminating repetitive validation logic in backend functions
- **Security & Reliability.**
 - Provides a first line of defense against malformed or malicious requests
 - Ensures data integrity before processing begins
- **Performance.**
 - Filters out invalid requests early in the request lifecycle
 - Reduces unnecessary backend processing and improves overall API performance

⌘ Step 1 : Creating DynamoDB table.

- Navigate to DynamoDB → [Create Table](#)
- Table Name : `customers`
- Partition key : `customer_id` (Number)
- Leave others to default and click on [create table](#)

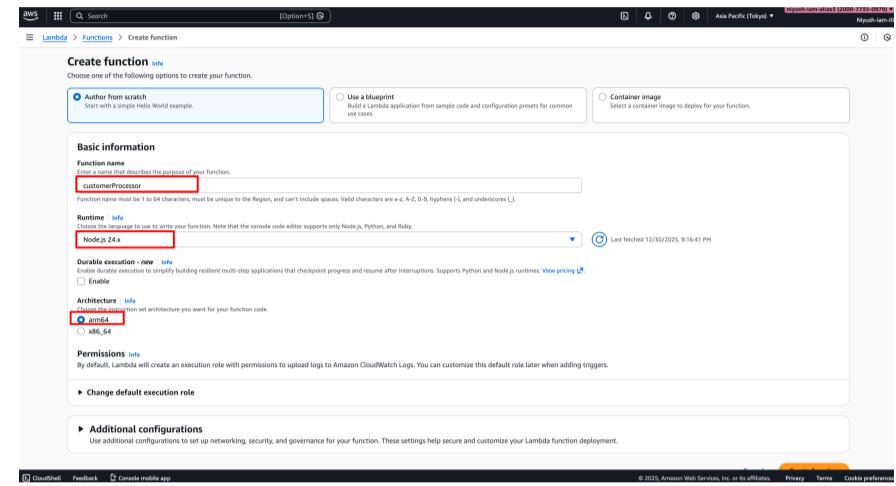
▼ Image : Creating DynamoDB Table

The screenshot shows the 'Create table' wizard in the AWS DynamoDB console. In the 'Table details' section, the table name is set to 'customers'. In the 'Partition key' section, the key name is 'customer_id' and the type is 'Number'. In the 'Table settings' section, the 'Default settings' radio button is selected, and the table class is set to 'DynamoDB Standard'. The 'Customize settings' option is also available.

⌘ Step 2 : Create Lambda Function

- Navigate to Lambda → [Create Function](#)
- Function name : `customerProcessor`
- Runtime : `Node.js`
- Architecture : `arm64`
- Finally [Create function](#)

▼ Image : Creating Lambda Function.



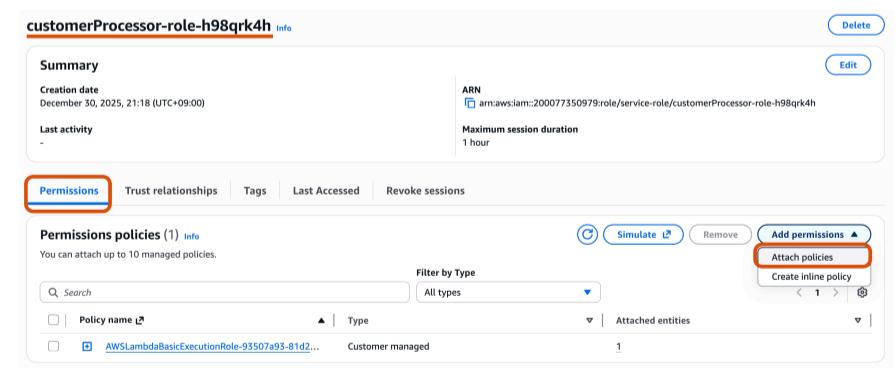
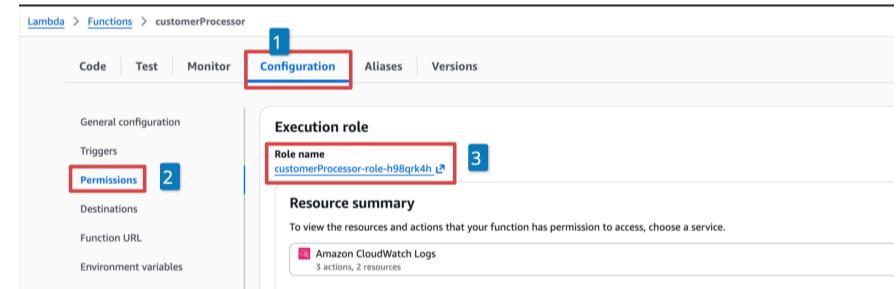
⌘ Step 2a : Adding DynamoDB permissions to Lambda Execution role.

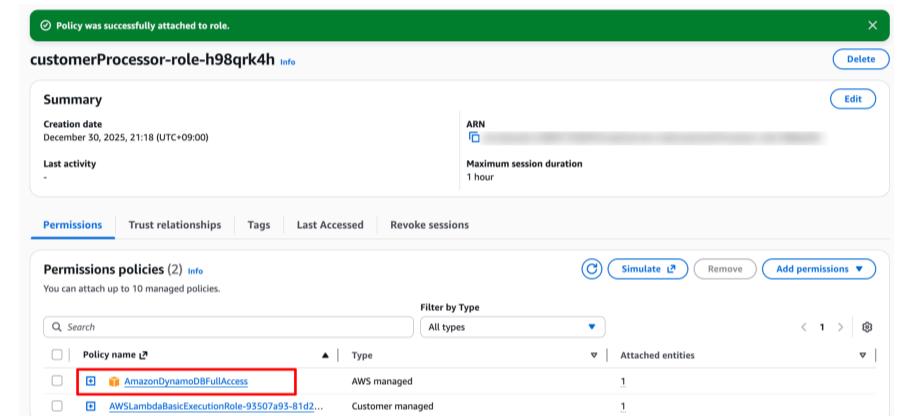
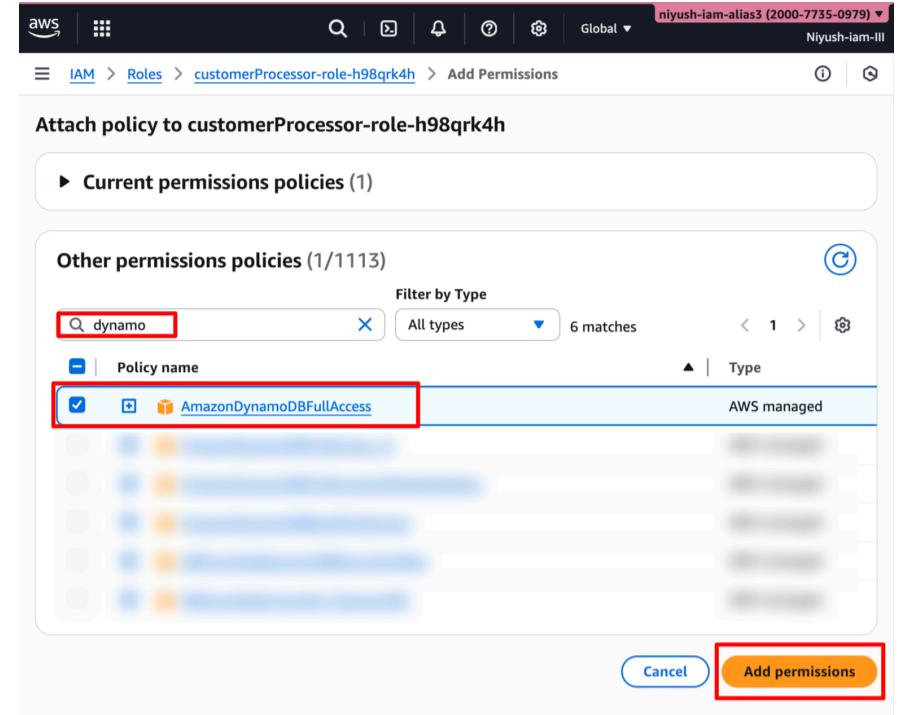
- In Lambda function page → [Configuration](#) → [Permission](#)
- execution role name under "Execution Role"
- In IAM [Add permissions](#) → [Attach policies](#)
- Look for : `AmazonDynamoDBFullAccess` and attach it.

⌘ Step 2b : Add the lambda code.

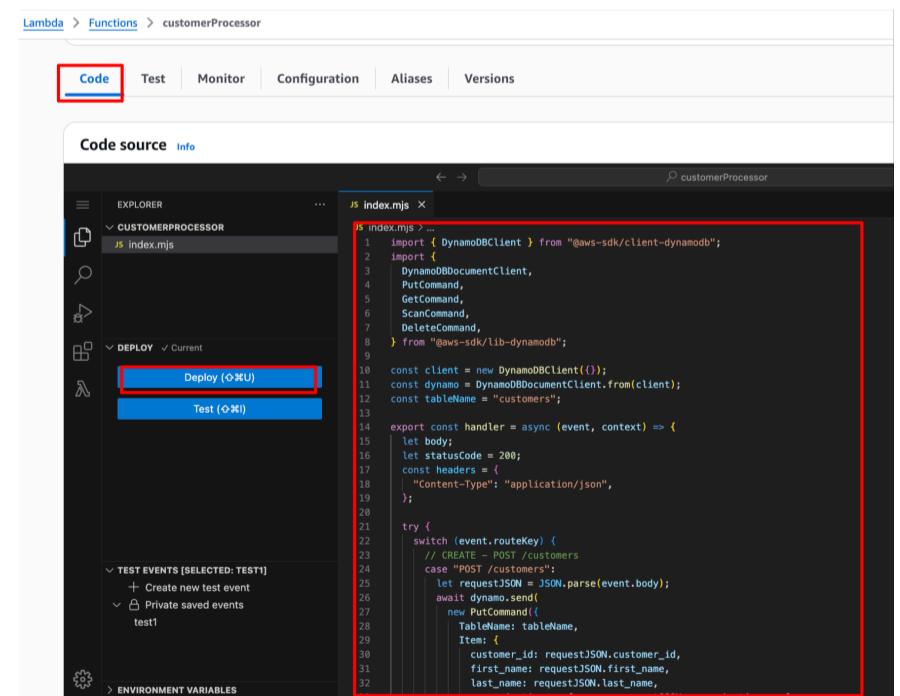
- In lambda , [code](#) section paste the code and hit [deploy](#) .

▼ Image : Adding DynamoDB permissions to Lambda





▼ Image : Test and Deploy the Code



▼ Node JS code below

```

import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import {
  DynamoDBDocumentClient,
  PutCommand,
  GetCommand,
  ScanCommand,
  DeleteCommand,
} from "@aws-sdk/lib-dynamodb";

const client = new DynamoDBClient({});
const dynamo = DynamoDBDocumentClient.from(client);
const tableName = "customers";

export const handler = async (event, context) => {
  let body;
  let statusCode = 200;
  const headers = {
    "Content-Type": "application/json",
  };

  try {
    // Get HTTP method and resource path
    const httpMethod = event.httpMethod;
    const resource = event.resource;
    const routeKey = `${httpMethod} ${resource}`;

    console.log("Route Key:", routeKey);
    console.log("Event:", JSON.stringify(event));

    switch (routeKey) {
      // CREATE - POST /customers
      case "POST /customers":
        let requestJSON = JSON.parse(event.body);
        await dynamo.send(
          new PutCommand({
            TableName: tableName,
            Item: {
              customer_id: requestJSON.customer_id,
              first_name: requestJSON.first_name,
              last_name: requestJSON.last_name,
              communication_preference: requestJSON.communication_preference,
              created_at: new Date().toISOString(),
            },
          })
        );
        body = `Customer ${requestJSON.customer_id} created successfully`;
        break;

      // READ ALL - GET /customers
      case "GET /customers":
        body = await dynamo.send(
          new ScanCommand({ TableName: tableName })
        );
        body = body.Items;
        break;
    }
  } catch (err) {
    console.error(err);
    statusCode = 500;
    body = `Internal Server Error: ${err.message}`;
  }
}


```

```

// READ ONE - GET /customers/{id}
case "GET /customers/{id}":
    body = await dynamo.send(
        new GetCommand({
            TableName: tableName,
            Key: {
                customer_id: parseInt(event.pathParameters.id),
            },
        })
    );
    body = body.Item;
    break;

// UPDATE - PUT /customers/{id}
case "PUT /customers/{id}":
    let updateJSON = JSON.parse(event.body);
    await dynamo.send(
        new PutCommand({
            TableName: tableName,
            Item: {
                customer_id: parseInt(event.pathParameters.id),
                first_name: updateJSON.first_name,
                last_name: updateJSON.last_name,
                communication_preference: updateJSON.communication_preference,
                created_at: new Date().toISOString(),
            },
        })
    );
    body = `Customer ${event.pathParameters.id} updated successfully`;
    break;

// DELETE - DELETE /customers/{id}
case "DELETE /customers/{id}":
    await dynamo.send(
        new DeleteCommand({
            TableName: tableName,
            Key: {
                customer_id: parseInt(event.pathParameters.id),
            },
        })
    );
    body = `Customer ${event.pathParameters.id} deleted successfully`;
    break;

default:
    throw new Error(`Unsupported route: "${routeKey}"`);
}

} catch (err) {
    statusCode = 400;
    body = err.message;
} finally {
    body = JSON.stringify(body);
}

return {
    statusCode,
    body,
    headers,
}

```

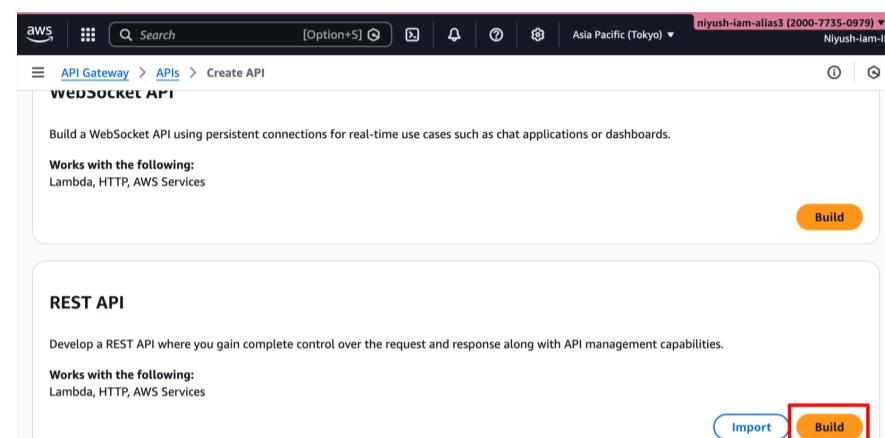
```
};  
};
```

⌘ Step 3 : Setting up the API GateWay.

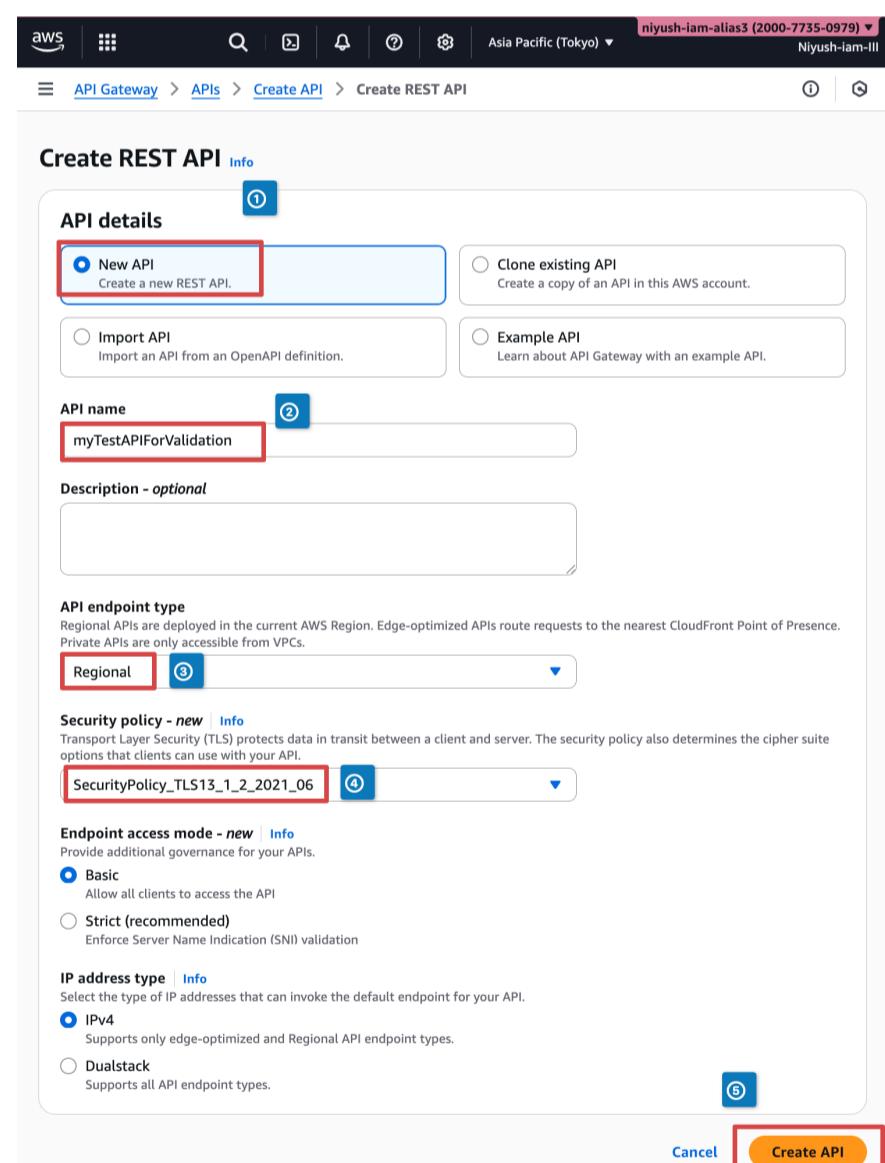
- Navigate to AWS Console ⇒ Head over to [API Gateway](#)
- Select [REST API](#) ⇒ click on [Build](#)
- Protocol : [REST](#) ⇒ [New API](#)
- API Name : [myTestAPIForValidation](#)
- Description : optional
- Endpoint Type : [Regional](#)
- Security Policy : [SecurityPolicy_TLS13_1_2_2021_6](#)

"Finally click on [Create API](#)"

▼ Image : REST API Creation



Creating the REST API



⌘ Step 4 : Create a Model.

- Go to API ⇒ from left nav panel [Model](#) ⇒ [Create Model](#)
- Model name : [mySampleModel](#)
- Content type : [application/json](#)

▼ Model Schema

▼ Image : Model Creation.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title" : "Customer Schema",
    "type" : "object",
    "properties": {
        "customer_id": {
            "type": "integer"
        },
        "first_name": {
            "type": "string"
        },
        "last_name": {
            "type": "string"
        },
        "communication_preference": {
            "enum": ["email", "sms", "voice_call"]
        }
    },
    "required": ["customer_id", "first_name", "last_name", "communication_preference"]
}
```

The screenshot shows the AWS API Gateway interface. In the top navigation bar, the path is: API Gateway > APIs > myTestAPIForValidation (tfrbr0p2qa) > Models. On the left sidebar, under the API section, the 'Models' tab is selected. In the main content area, there is a heading 'Models (0)' with three buttons: 'Delete', 'Edit', and 'Update documentation'. Below this, a sub-section titled 'API: myTestAPIForValidation' lists 'Resources', 'Stages', 'Authorizers', 'Gateway responses', and 'Models'. The 'Models' link is highlighted with a red box. A note says 'Use models to define the format for the body of different requests and responses used by your API.' At the bottom right of the main area, there is a 'Create model' button.

Creating model

This screenshot shows the 'Create' dialog for a new model. The top part has fields for 'Name' (set to 'mySampleModel') and 'Content type' (set to 'application/json'). Below this is a 'Description - optional' field with a placeholder. The main area is titled 'Model schema' and contains a JSON schema with line numbers from 1 to 19. The entire schema is highlighted with a red box. At the bottom right of the dialog, there are 'Cancel' and 'Create' buttons, with 'Create' also highlighted with a red box.

Defining the schema for the model

- Click on **Create** to create the model.

⌘ Step 5 : Create a Resource & Enable Model for the API.

- Go to API ⇒ click **Resources**

▼ Resource creation

- Select the root resource **/**

- Click Create Resource**

- Resource name: **customers**

- Resource path: **/customers**

- CORS**

- Click Create Resource**

- Select **/customers** resource

- Click Create Method**

- Select **POST**

- Integration type: **Lambda Function**

- Toggle Lambda Proxy Integration ON**

- Lambda Function: **customerProcessor**

- Click Save**

The screenshot shows the 'Create resource' dialog. At the top, it says 'Create resource'. Under 'Resource details', there is a radio button for 'Proxy resource' which is selected. A note says 'Proxy resources handle requests to all sub-resources. To create a proxy resource use a path parameter that ends with a plus sign, for example {proxy+}.' Below this are fields for 'Resource path' (set to '/') and 'Resource name' (set to 'customers'). There is also a checked checkbox for 'CORS (Cross Origin Resource Sharing) Info'. At the bottom right of the dialog, there are 'Cancel' and 'Create resource' buttons, with 'Create resource' also highlighted with a red box.

Creating resource

- Click **OK** when prompted to add permissions

 Create method for all i.e. **POST** **PUT** **GET** **DELETE**

Resources	/customers	/customers/{id}
	GET and POST	PUT, DELETE, GET

⌘ Step 5a : Set Up Request Validator

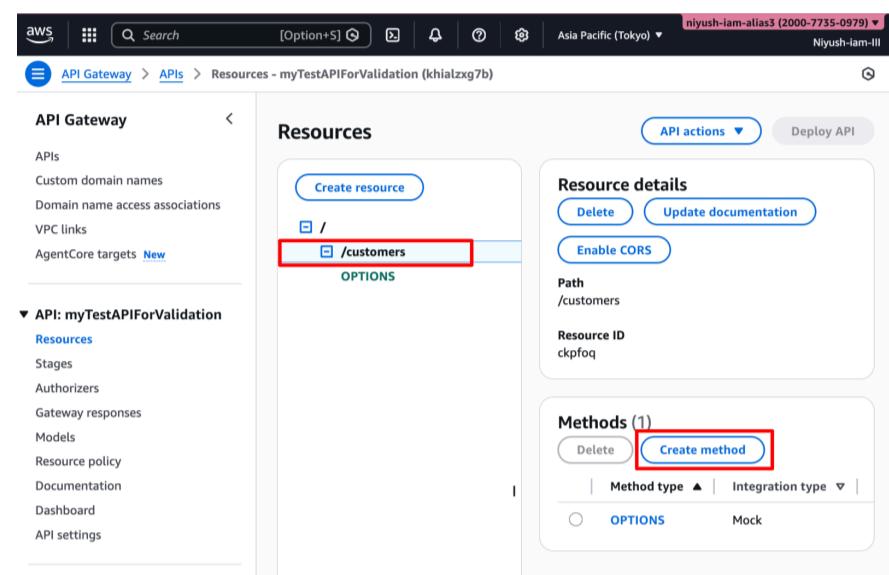
- Click **Request Validators**
- Click **Create Request Validator**
- Name: **ValidateCustomerBody**
- Check: **Validate request body**
- Uncheck: "Validate request parameters"
- Click **Create**

⌘ Step 5b : Apply Validator to POST Method

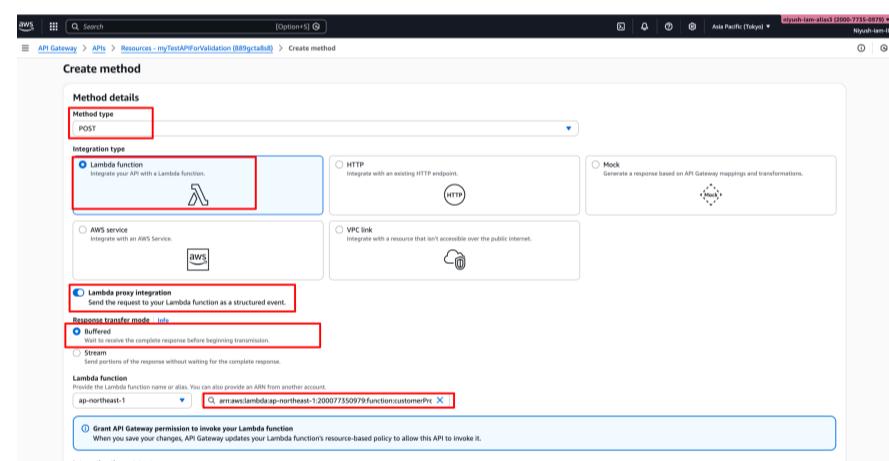
- Go to **Resources** → **/customers** → **POST**
- Click **Method Request** ⇒ **Edit**
- Scroll down to **Request Validator** dropdown
- Select **ValidateCustomerBody**
- Scroll to **Request Body** section
- Click **Add Model**
 - Content-Type: **application/json**
 - Model name: **mySampleModel**

▼ ABOUT OPTIONS and Request Validator

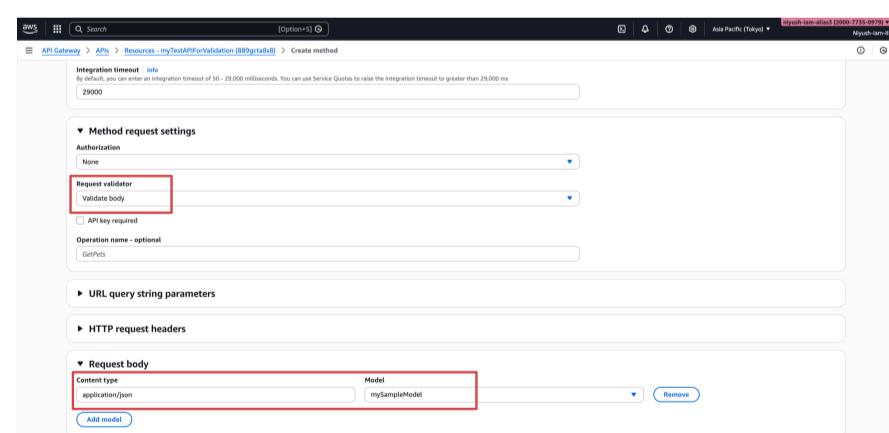
▼ POST Method Creation



Creating POST Method



Don't forget to toggle Lambda proxy integration



Request Validator : **validate body** Request Body : **application/json**,
mySampleModel



OPTIONS method:

- Used for **CORS preflight requests**
- Automatically sent by browsers before actual requests
- Has **no request body** to validate
- Should return quickly with headers only

What OPTIONS does:

- Tells client what methods are allowed (GET, POST, PUT, DELETE)
- Tells client what headers are accepted
- Returns 200 OK with CORS headers

▼ GET Method Creation

Request Validator and Request Body not required

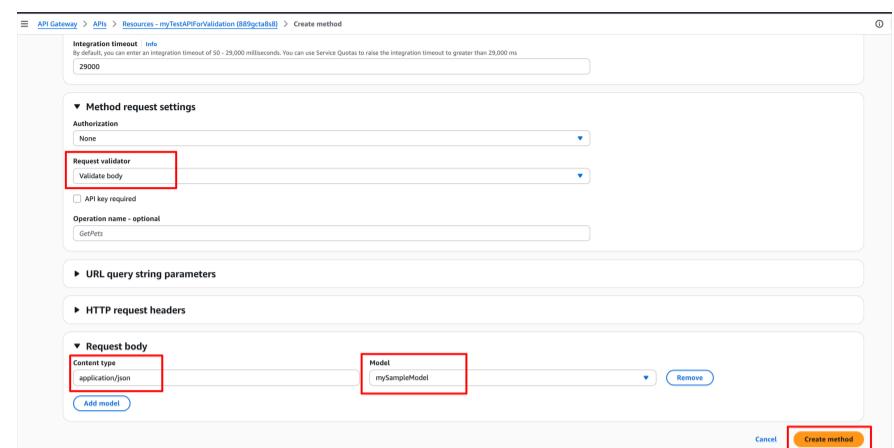
Method	resource	Request Validator	Request Body
OPTIONS	/customers	Not Required	Not required
POST	/customers	Required	Required
GET	/customers	NO (no body in GET)	NO (no body in GET)
GET single	/customer/{id}	NO (no body in GET)	NO (no body in GET)
PUT	/customers/{id}	Required	Required
DELETE	/customers/{id}	NO (no body in DELETE)	NO (no body in GET)

▼ DELETE Method Creation

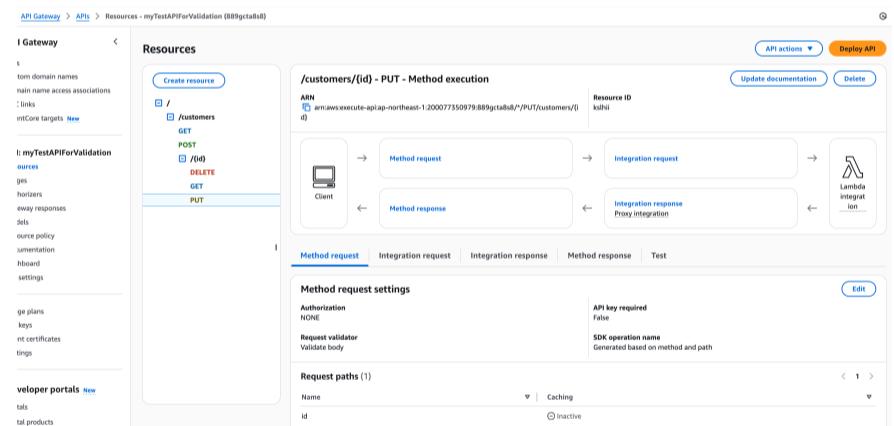
Request Validator and Request Body not required

▼ PUT Method Creation

Request Validator and Request Body required



▼ Final Product once all Method are created



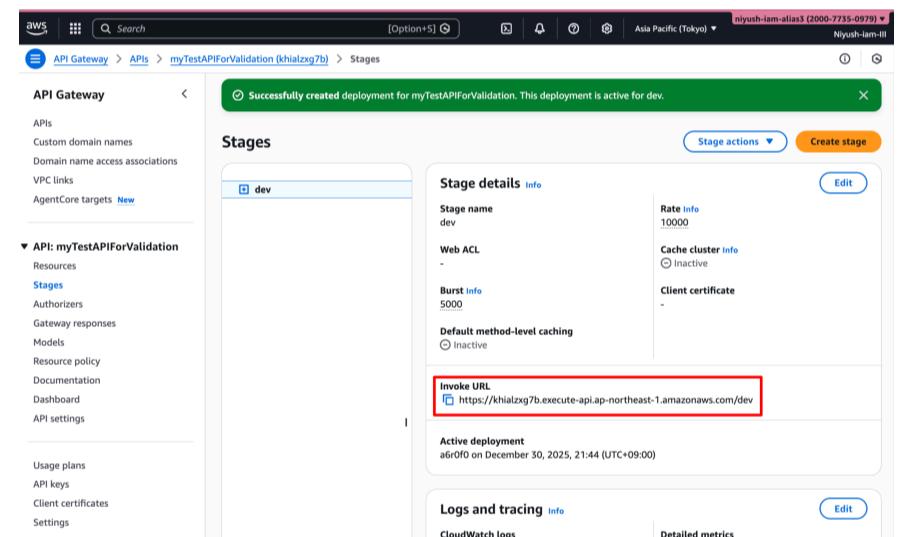
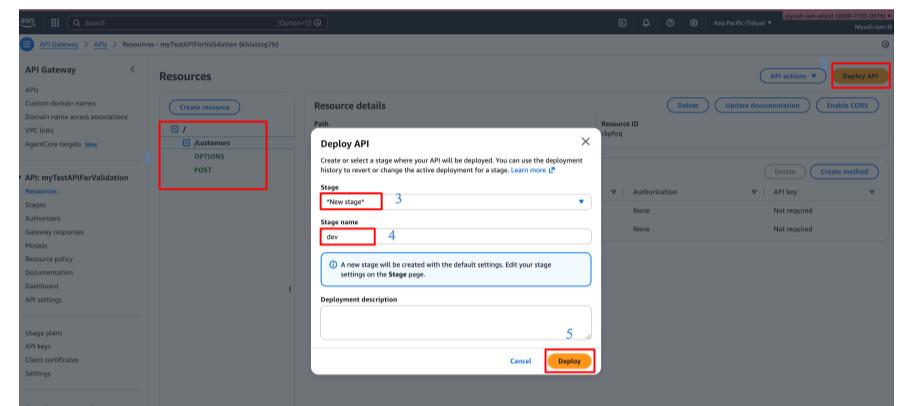
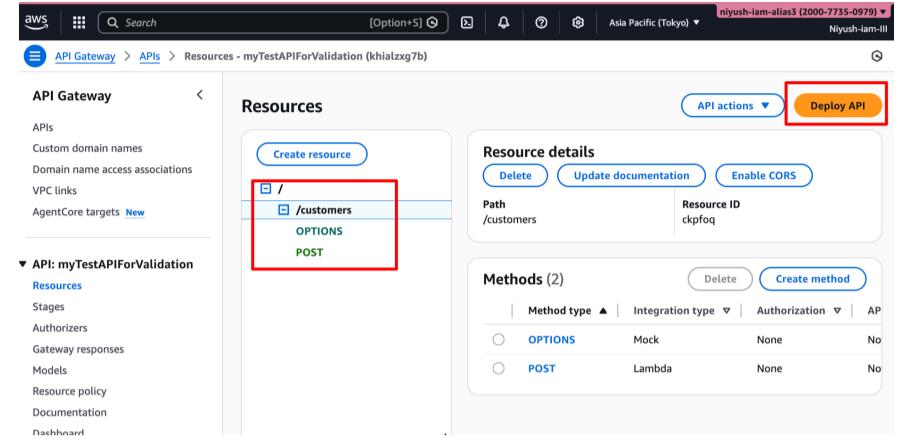
⌘ Step 6 : Deploy the API.

- Resource ⇒ / ⇒ POST ⇒ Actions ⇒ Deploy API.
- Deployment Stage : NewStage
- Stage name : test

Finally click on Deploy

Head over to Stages ⇒ Click on Deployment ⇒ Copy the Invoke URL to test the API endpoint.

▼ Image : Deploying the API



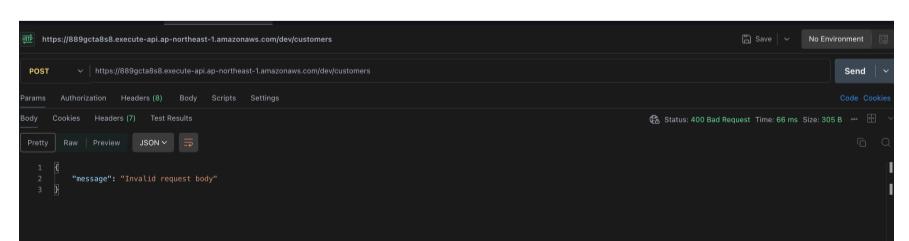
⌘ Step 7 : Test and Validate the API Endpoint with Request Payload.

Open [Postman](#) and Click on [New Request](#) ⇒ [POST](#) request

Test #1 : No Request Body payload.

- in [POST](#) request enter the [Invoke URL](#)
- Click on [Send](#)
- you should see a [Status : 400 Bad Request](#) as response.

▼ Image : Test #1 No Request Body payload



Expected Response.

Status: 400 Bad Request

```
{
  "message": "Invalid request body"
}
```

Test #2 : Wrong Schema Parameters.

As per the defined JSON Schema `communication_preference` can only be `[email, sms, voice_call]`

- In Postman, with the same Invoke URL ⇒ `POST`
- Body: use the below test pattern

```
{
  "customer_id": "122",
  "first_name": "John",
  "last_name": "Doe",
  "communication_preference": "Phone_Call"
}
```

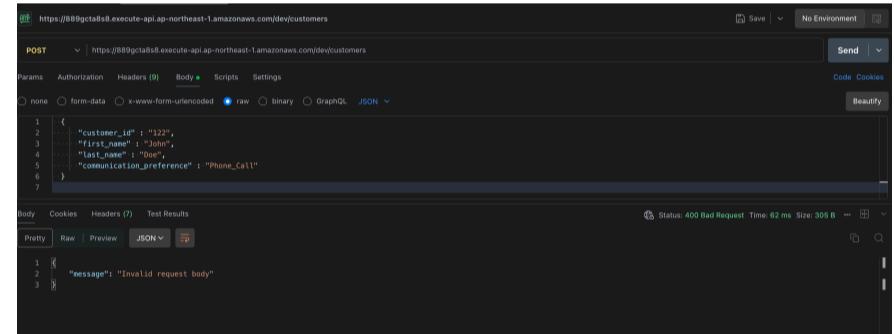
▼ Image : Test #2 Wrong Schema parameters

This will return an error since we have not define Phone_call option in our JSON Schema.

Expected Response.

Status: 400 Bad Request

```
{
  "message": "Invalid request body",
  "errors": [
    "Invalid enum value. Must be one of: email, sms, voice_call"
  ]
}
```



Test #3 : Correct Schema Parameters.

In postman follow the same steps taken above for test 1 and 2.

- New Request
- Method : `POST`
- Copy and Paste the API Gateway's Invoke URL.
- In the body section paste the below code.

```
{
  "customer_id": 123,
  "first_name": "John",
  "last_name": "Doe",
  "communication_preference": "email"
}
```

- finally click on `Send`

Expected Response.

- This time you should see Status : 200 OK

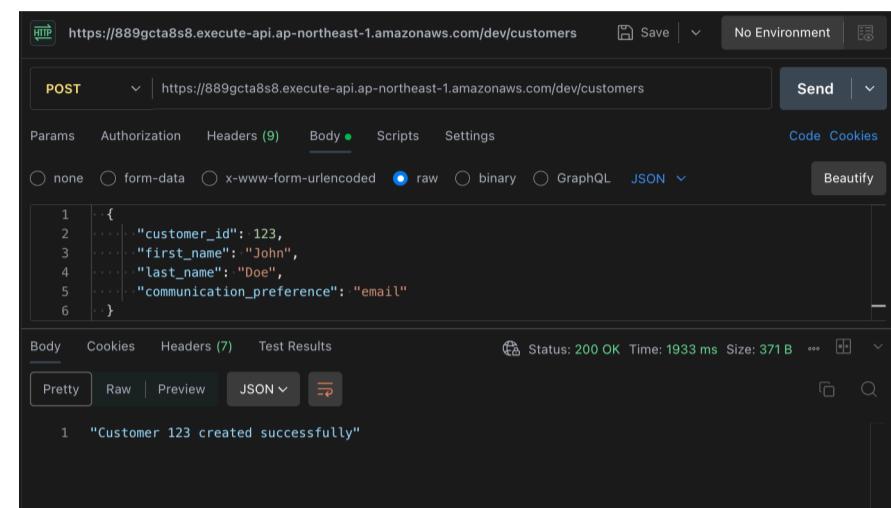
Test #4: Missing Required Field

The schema requires all four fields. Let's test what happens when we omit one.

- Method: POST
- Invoke URL: (same as before)
- Body:

```
{
  "customer_id": 123,
  "first_name": "John",
  "last_name": "Doe"
  // Missing: communication_preference
}
```

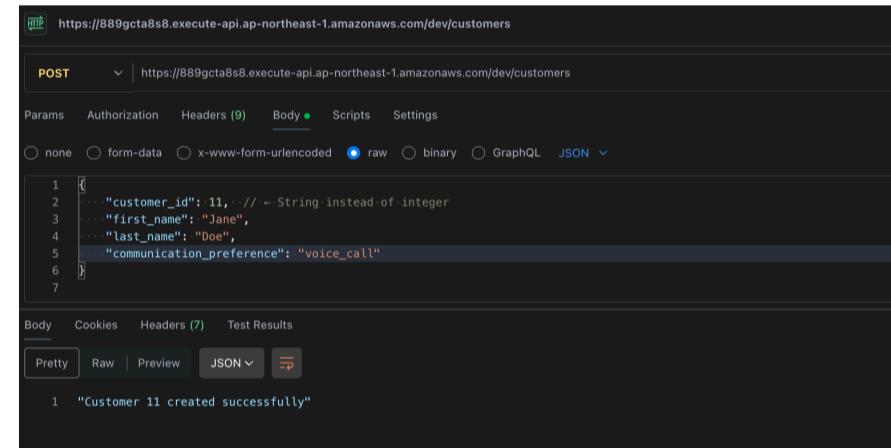
▼ Image : Test #3 Correct Schema parameters



Expected Response :

Status: 400 Bad Request

```
{
  "message": "Invalid request body",
  "errors": [
    "Missing required property: communication_preference"
  ]
}
```



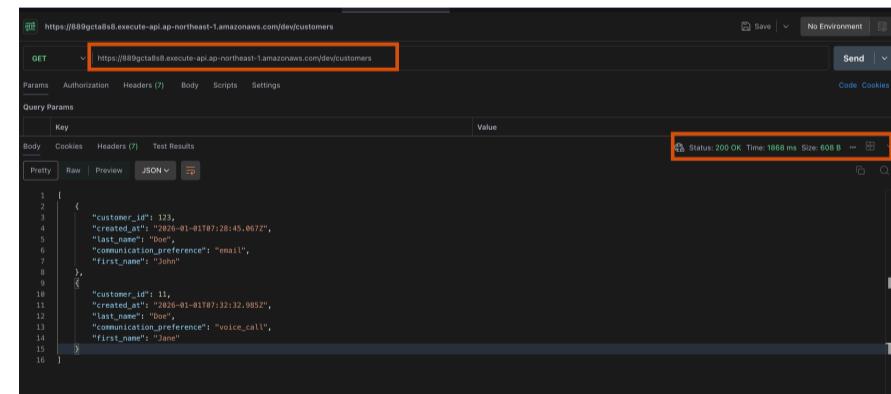
Test #5 : Wrong Data Type.

customer_id should be an integer, not a string.

- Method: POST
- Invoke URL: (same as before)
- Body:

```
{
  "customer_id": "123", // ← String instead of integer
  "first_name": "John",
  "last_name": "Doe",
  "communication_preference": "email"
}
```

▼ GET all items

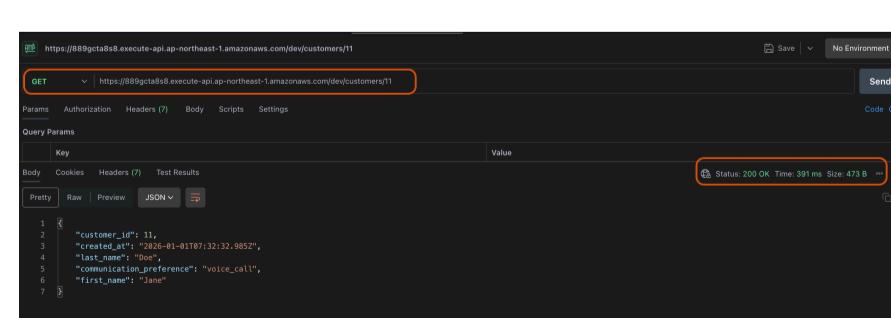


Expected Response :

Status: 400 Bad Request

```
{
  "message": "Invalid request body",
  "errors": [
    "Invalid type. Expected: integer, but got: string"
  ]
}
```

▼ GET Single Item



▼ Creating Items with POST

A screenshot of the Postman interface. The URL is `https://889gcta8s8.execute-api.ap-northeast-1.amazonaws.com/dev/customers`. The method is set to `POST`. The body contains the following JSON:

```
1 {  
2   "customer_id": 12,  
3   "first_name": "SpongeBob",  
4   "last_name": "SquarePants",  
5   "communication_preference": "voice_call"  
6 }  
7
```

The response status is `200 OK`, time `304 ms`, size `370 B`. The response body is: "Customer 12 created successfully".

▼ Updating the Item in DB with PUT

Currently in DB the value of 11 is Jane Doe

Table: customers - Items returned (4)
Scan started on January 01, 2026, 16:51:00

	customer_id (Number)	communication_preference	created_at	first_name	last_name
	11	voice_call	2026-01-01T07:32:32.985Z	Jane	Doe
	123	email	2026-01-01T07:28:45.067Z	John	Doe
	13	voice_call	2026-01-01T07:50:28.272Z	Patrick	Star
	12	voice_call	2026-01-01T07:49:36.612Z	SpongeBob	SquarePants

After using PUT

A screenshot of the Postman interface. The URL is `https://889gcta8s8.execute-api.ap-northeast-1.amazonaws.com/dev/customers/11`. The method is set to `PUT`. The body contains the following JSON:

```
1 {  
2   "customer_id": 11,  
3   "first_name": "Eugene",  
4   "last_name": "Krabs",  
5   "communication_preference": "sms"  
6 }  
7
```

The response status is `200 OK`, time `312 ms`, size `370 B`. The response body is: "Customer 11 updated successfully".

The value of customer_id 11 is now updated successfully

Table: customers - Items returned (4)
Scan started on January 01, 2026, 16:53:33

	customer_id (Number)	communication_preference	created_at	first_name	last_name
	11	sms	2026-01-01T07:53:26.471Z	Eugene	Krabs
	123	email	2026-01-01T07:28:45.067Z	John	Doe
	13	voice_call	2026-01-01T07:50:28.272Z	Patrick	Star
	12	voice_call	2026-01-01T07:49:36.612Z	SpongeBob	SquarePants

▼ Removing a single Item with DELETE

Items currently in database before deletion.

Table: customers - Items returned (4)						
Scan started on January 01, 2026, 16:53:33						
	customer_id (Number)	communication_preference	created_at	first_name	last_name	
1	11	sms	2026-01-01T07:53:26.471Z	Eugene	Krabs	
2	123	email	2026-01-01T07:28:45.067Z	John	Doe	
3	13	voice_call	2026-01-01T07:50:28.272Z	Patrick	Star	
4	12	voice_call	2026-01-01T07:49:36.612Z	SpongeBob	SquarePants	

DELETE Method.

The screenshot shows the Postman interface with a DELETE request to the specified URL. The response status is 200 OK, and the message "Customer 123 deleted successfully" is displayed.

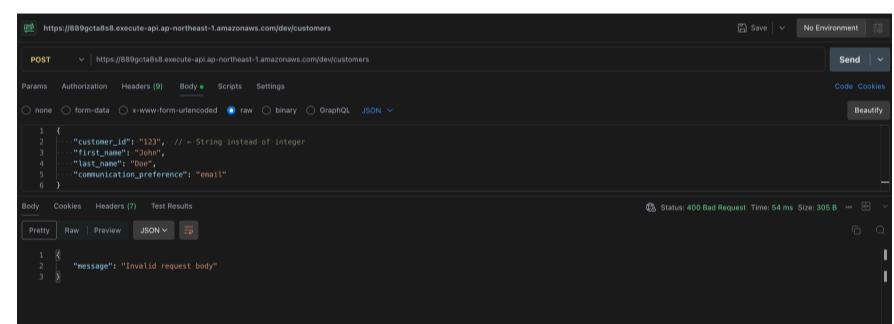
After deleting and checking the Database.

Table: customers - Items returned (3)						
Scan started on January 01, 2026, 16:55:28						
	customer_id (Number)	communication_preference	created_at	first_name	last_name	
1	11	sms	2026-01-01T07:53:26.471Z	Eugene	Krabs	
2	13	voice_call	2026-01-01T07:50:28.272Z	Patrick	Star	
3	12	voice_call	2026-01-01T07:49:36.612Z	SpongeBob	SquarePants	

▼ Image : Test #4 Missing parameters

The screenshot shows the Postman interface with a POST request to the specified URL. The request body is a JSON object with customer_id: 123, first_name: 'SpongeBob', and last_name: 'SquarePants'. The communication_preference field is missing. The response status is 400 Bad Request, and the message "Invalid request body" is displayed.

▼ Image : Test #5 Wrong data Type



The screenshot shows the AWS Lambda Test API interface. A POST request is being made to the URL `https://889gctas8d.execute-api.ap-northeast-1.amazonaws.com/dev/customers`. The request body is a JSON object:

```
1 {
2   "customer_id": "123", // - String instead of integer
3   "first_name": "John",
4   "last_name": "Doe",
5   "communication_preference": "email"
6 }
```

The response status is 400 Bad Request, with the message "Invalid request body".

⌘ Troubleshooting

Issue: Getting 403 Forbidden

Solution: Make sure you're using the correct Invoke URL from the Stages section.

Issue: Getting 404 Not Found

Solution:

- Verify the API is deployed
- Check the stage name matches (should be `test`)
- Ensure you're using the correct HTTP method (POST)

Issue: Validation not working (accepting invalid data)

Solution:

- Go back to Step 3 and verify "Enable Validation Body in Request Validation" is checked
- Confirm the model `mySampleModel` is selected
- Redeploy the API after making changes

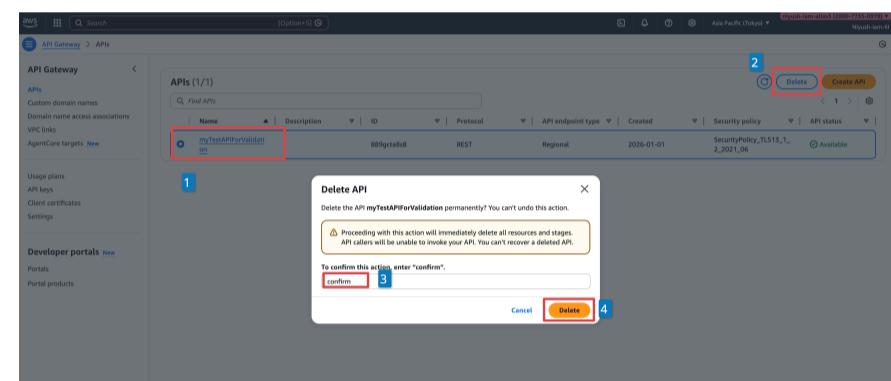
⌘ Step 8 : Deletion Time.

After testing, it's important to delete your API Gateway to avoid unnecessary AWS charges.

Delete the API Gateway

- Navigate to **AWS Console** ⇒ **API Gateway**
- From the left panel, select your API: `myTestAPIForValidation`
- Click on **Actions** (top-right dropdown)
- Select **Delete API**
- A confirmation dialog will appear asking: "Are you sure you want to delete this API?"
- Click **Delete** to confirm

▼ Delete the API Gateway



Delete the Model (Optional)

If you want to delete just the model without deleting the entire API:

- Navigate to **API Gateway** ⇒ Your API: `myTestAPIForValidation`
- From the left nav panel, click **Models**
- Select your model: `mySampleModel`
- Click **Delete Model**
- Confirm the deletion

Note: If you delete the API, the model is automatically deleted with it.

Delete APIs via AWS CLI (Optional)

If you prefer using the command line:

```
# List all APIs to find the API ID
aws apigateway get-rest-apis

# OR
aws apigateway get-rest-apis | grep -C 5 "YOUR_API_NAME"

# Delete the API (replace API_ID with your actual API ID)
aws apigateway delete-rest-api --rest-api-id API_ID
```

▼ Delete the API Gateway via terminal

```
# Example:  
aws apigateway delete-rest-api --rest-api-id 1a2b3c4d5  
e
```

Expected Output:

(No output means successful deletion)

Delete the DynamoDB

```
Last login: Mon Dec 29 12:42:29 on ttys000  
niyushbjr@Niyushs-MacBook-Air ~ % aws apigateway get-rest-apis  
{  
  "items": [  
    {  
      "id": "889gcta8s8",  
      "name": "myTestAPIForValidation",  
      "createdAt": "2026-01-01T16:21:08+09:00",  
      "apiKeySource": "HEADER",  
      "endpointConfiguration": {  
        "types": [  
          "REGIONAL"  
        ],  
        "ipAddressType": "ipv4"  
      },  
      "disableExecuteApiEndpoint": false,  
      "rootResourceId": "p6uenq9y5b"  
    }  
  ]  
}  
niyushbjr@Niyushs-MacBook-Air ~ % aws apigateway delete-rest-api --rest-api-id 889gcta8s8  
No output means successfully deleted.
```

Deleting DynamoDB via AWS Console.

- Head over to DynamoDB → select the table
- Actions → Delete → Type in **confirm** → and Finally click on **Delete**

Deleting DynamoDB via Terminal or AWS CLI

use the following command.

```
aws dynamodb list-tables  
  
aws dynamodb delete-table \  
--table-name YOUR_TABLE_NAME  
  
aws dynamodb describe-table --table-name customers
```

Delete the Lambda Function and Role associated with it.

Deleting Lambda Function via Console

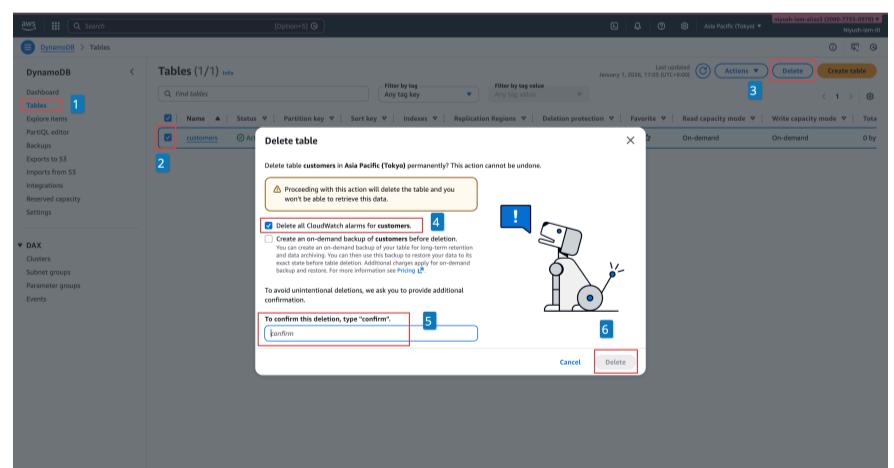
- Lambda → select the function → Actions → Delete

Deleting Lambda Function via Terminal or AWS CLI

use the following command.

```
aws lambda list-functions  
  
aws lambda delete-function \  
--function-name YOUR_FUNCTION_NAME
```

▼ Deleting dynamoDB via console



Deleting via AWS CLI

Delete the Role associated with it Lambda Function

- Head over to IAM → Roles → select the function role → Delete

```
niyushbjr@Niyushs-MacBook-Air ~ % aws dynamodb list-tables
{
    "TableNames": [
        "customers"
    ]
}
niyushbjr@Niyushs-MacBook-Air ~ % aws dynamodb delete-table --table-name customers
{
    "TableDescription": {
        "TableName": "customers",
        "TableStatus": "DELETING",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 0,
            "WriteCapacityUnits": 0
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:ap-northeast-1:200077350979:table/customers",
        "TableId": "913ebab8-41a1-b2c0-d9dc0d060089",
        "BillingModeSummary": {
            "BillingMode": "PAY_PER_REQUEST",
            "LastUpdateToPayPerRequestDateTime": "2026-01-01T16:18:15.437000+09:00"
        },
        "TableClassSummary": {
            "TableClass": "STANDARD"
        },
        "DeletionProtectionEnabled": false
    }
}
niyushbjr@Niyushs-MacBook-Air ~ % aws dynamodb describe-table --table-name customers
An error occurred (ResourceNotFoundException) when calling the DescribeTable operation: Requested resource not found: Table: customers not found
niyushbjr@Niyushs-MacBook-Air ~ %
```

Delete the Role associated with it Lambda Function

Deleting Log Group via Console

- Headover to Cloudwatch → select Log Management → /aws/... log
- Actions → Delete log groups(s)

▼ Deleting dynamoDB via Terminal

```
niyushbjr@Niyushs-MacBook-Air ~ % aws dynamodb list-tables
{
    "TableNames": [
        "customers"
    ]
}
niyushbjr@Niyushs-MacBook-Air ~ % aws dynamodb delete-table --table-name customers
{
    "TableDescription": {
        "TableName": "customers",
        "TableStatus": "DELETING",
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 0,
            "WriteCapacityUnits": 0
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:ap-northeast-1:200077350979:table/customers",
        "TableId": "913ebab8-41a1-b2c0-d9dc0d060089",
        "BillingModeSummary": {
            "BillingMode": "PAY_PER_REQUEST",
            "LastUpdateToPayPerRequestDateTime": "2026-01-01T16:18:15.437000+09:00"
        },
        "TableClassSummary": {
            "TableClass": "STANDARD"
        },
        "DeletionProtectionEnabled": false
    }
}
niyushbjr@Niyushs-MacBook-Air ~ % aws dynamodb describe-table --table-name customers
An error occurred (ResourceNotFoundException) when calling the DescribeTable operation: Requested resource not found: Table: customers not found
niyushbjr@Niyushs-MacBook-Air ~ %
```

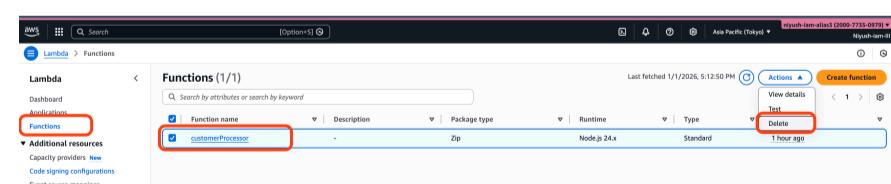
Deleting Log Group via AWS CLI

```
aws logs describe-log-groups
```

```
aws logs delete-log-group -- log-group-name NAME_OF_LOG_GROUP
```

```
aws logs describe-log-groups
```

▼ Deleting the lambda



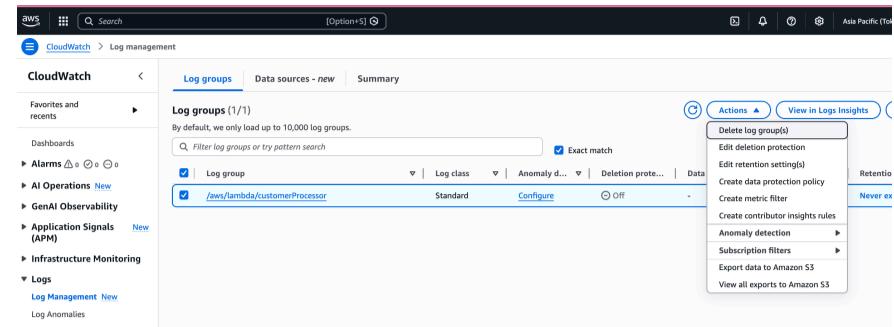
▼ Deleting Lamda via AWS CLI

```
iniyushbjr@Niyushs-MacBook-Air ~ % aws lambda list-functions
{
    "Functions": [
        {
            "FunctionName": "customerProcessor",
            "FunctionArn": "arn:aws:lambda:ap-northeast-1:200077350979:function:customerProcessor",
            "Runtime": "nodejs24.x",
            "Role": "arn:aws:iam::200077350979:role/service-role/customerProcessor-role-cdd9sak0",
            "Handler": "index.handler",
            "CodeSize": 994,
            "Description": "",
            "Timeout": 3,
            "MemorySize": 128,
            "LastModified": "2026-01-01T07:20:29.000+0000",
            "CodeSha256": "+CULcdWwp4cxXd6pQV1VXws2ybUo4SkGhA3set03gg=",
            "Version": "$LATEST",
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "RevisionId": "834c8080-657f-429b-abea-6f5679bae1e1",
            "PackageType": "Zip",
            "Architectures": [
                "arm64"
            ],
            "EphemeralStorage": {
                "Size": 512
            },
            "SnapStart": {
                "ApplyOn": "None",
                "OptimizationStatus": "Off"
            },
            "LoggingConfig": {
                "LogFormat": "Text",
                "LogGroup": "/aws/lambda/customerProcessor"
            }
        }
    ]
}
```

▼ Delete the Role assigned to Lambda

The screenshot shows the AWS IAM Roles page. On the left, there's a sidebar with 'Identity and Access Management (IAM)' and a 'Roles' section highlighted with a red box. The main area shows a table of roles. One row is selected, with its details shown in a modal at the bottom. The modal title is 'customerProcessor-role-cdd9sak0' and it says 'AWS Service: lambda'. A red box highlights the 'Delete' button in the top right corner of the modal.

▼ Delete the Cloudwatch log via console



▼ Delete the Cloudwatch log via AWS CLI

```
niyushbjr@Niyushs-MacBook-Air ~ % aws logs describe-log-groups
{
    "logGroups": [
        {
            "logGroupName": "/aws/lambda/customerProcessor",
            "creationTime": 1767252029932,
            "metricFilterCount": 0,
            "arn": "arn:aws:logs:ap-northeast-1:200077350979:log-group:/aws/lambda/customerProcessor:*",
            "storedBytes": 0,
            "logGroupClass": "STANDARD",
            "logGroupArn": "arn:aws:logs:ap-northeast-1:200077350979:log-group:/aws/lambda/customerProcessor"
        }
    ]
}
```

```
niyushbjr@Niyushs-MacBook-Air ~ % aws logs delete-log-group --log-group-name /aws/lambda/customerProcessor
niyushbjr@Niyushs-MacBook-Air ~ % aws logs describe-log-groups
{
    "logGroups": []
}
niyushbjr@Niyushs-MacBook-Air ~ %
```

⌘ Key Takeaways

✓ What This Demonstrates:

- How to create a JSON Schema in API Gateway
- How API Gateway validates requests **before** sending to backend
- How to catch invalid data early (saves Lambda costs)
- How validation errors are returned to the client

✓ Why This Matters:

- Prevents malformed data from reaching your backend
- Reduces backend processing for invalid requests
- Provides clear error messages to API consumers
- Improves API reliability and security

And that's a wrap