



CRUD Operations Website Using API GATEWAY

Html and JavaScript code

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>API Gateway CRUD Operations with CORS</title>
  <script>
    // Replace with your API Gateway URLs for each Lambda function

    // Function to perform a GET request
    async function getData() {
      const AppName = document.getElementById('get-appliance-name').value;
      console.log((AppName))

      if (!AppName) {
        alert('Please enter an appliance name.');
        return;
      }
    }
  </script>
</head>
<body>
  <input type="text" id="get-appliance-name" placeholder="Appliance Name">
  <button type="button" onclick="getData();">Get Appliance Data</button>
</body>
</html>
```

```

try {
    const response = await fetch(`https://i1qrnm2fgb.execute-api.us-east-1.amazonaws.com/prod/get`)
    method: 'GET',
    headers: {
        'Content-Type': 'application/json',
    }
});
const data = await response.json();
document.getElementById('output').textContent = JSON.stringify(data);
} catch (error) {
    console.error('Error fetching data:', error);
    document.getElementById('output').textContent = 'Error fetching data';
}
}

// Function to perform a POST request
async function postData() {
    const data = {
        AppName: document.getElementById('post-name').value,
        Status: document.getElementById('post-status').value
    };
    console.log(data)

    try {
        const response = await fetch(`https://i1qrnm2fgb.execute-api.us-east-1.amazonaws.com/prod/post`)
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
        },
        body: JSON.stringify(data)
    });
    const result = await response.json();
    document.getElementById('output').textContent = JSON.stringify(result);
} catch (error) {
    console.error('Error posting data:', error);
    document.getElementById('output').textContent = 'Error posting data';
}
}

// // Function to perform a PUT request by using one pathparameters
// async function putData() {
//     const id = document.getElementById('put-name').value;
//     const data = {
//         Status: document.getElementById('put-status').value
//     };

```

```

//      try {
//          const response = await fetch(` ${apiPutUrl}/ ${id} `, {
//              method: 'PUT',
//              headers: {
//                  'Content-Type': 'application/json',
//              },
//              body: JSON.stringify(data)
//          });
//          const result = await response.json();
//          document.getElementById('output').textContent = JSON.stringify(result);
//      } catch (error) {
//          console.error('Error updating data:', error);
//          document.getElementById('output').textContent = 'Error updating data';
//      }
// }

// Function to perform a PUT request
async function putData() {
    const data = {
        AppName: document.getElementById('put-name').value,
        Status: document.getElementById('put-status').value
    };
    console.log(data)
    console.log(JSON.stringify(data))
    console.log(data.AppName,data.Status)

    try {
        const response = await fetch(`https://i1qrnm2fgb.execute-api.us-east-1.amazonaws.com/Prod/applications/${id}`, {
            method: 'PUT',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify(data)
        });
        const result = await response.json();
        document.getElementById('output').textContent = JSON.stringify(result);
    } catch (error) {
        console.error('Error updating data:', error);
        document.getElementById('output').textContent = 'Error updating data';
    }
}

// Function to perform a DELETE request
async function deleteData() {

```

```

const AppName = document.getElementById('delete-id').value;
console.log(AppName)

try {
    const response = await fetch(`https://i1qrnm2fgb.execute-api.us-east-1.amazonaws.com/prod/applications/${AppName}`)
        .method: 'DELETE',
        headers: {
            'Content-Type': 'application/json',
        }
    });
    const result = await response.json();
    document.getElementById('output').textContent = JSON.stringify(result);
} catch (error) {
    console.error('Error deleting data:', error);
    document.getElementById('output').textContent = 'Error deleting data';
}
}
</script>
</head>
<body>
    <h1>API Gateway CRUD Operations with CORS</h1>

    <h2>GET Data</h2>
    <label for="get-appliance-name">Appliance Name:</label>
    <input type="text" id="get-appliance-name">
    <button onclick="getData()">Fetch Data</button>
    <pre id="output"></pre>

    <h2>POST Data</h2>
    <label for="post-name">Name:</label>
    <input type="text" id="post-name">
    <br>
    <label for="post-value">Status:</label>
    <input type="text" id="post-status">
    <br>
    <button onclick="postData()">Post Data</button>

    <h2>PUT Data</h2>
    <!-- <label for="put-id">ID:</label>
    <input type="text" id="put-id"> -->
    <!-- <br> -->
    <label for="put-name">Name:</label>
    <input type="text" id="put-name">
    <br>
    <label for="put-value">Status:</label>
    <input type="text" id="put-status">

```

```
<br>
<button onclick="putData()">Update Data</button>

<h2>DELETE Data</h2>
<label for="delete-id">AppName:</label>
<input type="text" id="delete-id">
<br>
<button onclick="deleteData()">Delete Data</button>
</body>
</html>
```

Explanation For the code



The Above html file is a sample website that's going to Perform ("GET"- "PUT"- "POST"") and "DELETE" API call operations using Four separate Lambda functions.

1. The html file contains inline JavaScript Elements that's going to perform the function of calling Api when every we click a button.
2. Code Contains four JavaScript Functions Named (getData(), postData(), putData(), deleteData()) to perform the respective Operations.

▼ **getData() - (GET)**

```
<h2>GET Data</h2>
<label for="get-appliance-name">Appliance Name:</label>
<input type="text" id="get-appliance-name">
<button onclick="getData()">Fetch Data</button>
<pre id="output"></pre>
```

1. When you input a value in the website for the "Appliance Name" label field it's going to get stored in the id="get-appliance-name"
2. Now when you click on "Fetch Data" button it's going to call the "getData()" function
- 3.

```
<button onclick="getData()">Fetch Data</button>
```

4. This Function call will call the getData()

5.

```
// Function to perform a GET request
async function getData() {
    const AppName = document.getElementById('get-appliance-name')
    console.log((AppName))

    if (!AppName) {
        alert('Please enter an appliance name.');
        return;
    }

    try {
        const response = await fetch(`https://i1qrnm2fgb.execute-api.us-east-1.amazonaws.com/execute-api/test-endpoint`)
        const data = await response.json()
        console.log(data)
    }
}
```

```

        });
        const data = await response.json();
        document.getElementById('output').textContent = JSON.stringify(data);
    } catch (error) {
        console.error('Error fetching data:', error);
        document.getElementById('output').textContent = 'Error fetching data';
    }
}

```

Function is going to take the input value given in the appliance name "label" and pass it to AppName const.

```
const AppName = document.getElementById('get-appliance-name').value;
```

Next we are triggering the Api as path Parameters, The AppName is injected into the url.

```
const response = await fetch(`https://i1qrnm2fgb.execute-api.us-east-1.amazonaws.com/PROD/GET/${AppName}`, {
    method: 'GET',
    headers: {
        'Content-Type': 'application/json',
    }
});
```

Let's break down the line of code you provided in detail to understand what it does and how it works:

Code Explanation

```
const response = await fetch(`https://i1qrnm2fgb.execute-api.us-east-1.amazonaws.com/PROD/GET/${AppName}`, {
    method: 'GET',
    headers: {
        'Content-Type': 'application/json',
    }
});
```

1. `const response = await fetch(...)`

- **fetch Function:**

- `fetch` is a built-in JavaScript function that makes network requests to servers. It's part of the Fetch API and is commonly used to send HTTP requests (GET, POST, PUT, DELETE, etc.) to web servers and receive responses.

- The `fetch` function returns a `Promise` that resolves to the `Response` object representing the response to the request.
- **`await` Keyword:**
 - The `await` keyword is used to wait for the `Promise` returned by `fetch` to be resolved (i.e., complete the network request). Using `await` ensures that the execution of the code pauses at this point until the `fetch` operation completes, which allows you to work with the response synchronously.
 - To use `await`, the surrounding function must be marked with the `async` keyword, making an asynchronous function.
- **`const response`:**
 - This line declares a constant variable `response` that will store the result of the `fetch` call once it completes.

2. URL in Backticks: ``https://i1qrnm2fgb.execute-api.us-east-1.amazonaws.com/PROD/GET/${AppName}``

- **URL Structure:**
 - The URL `https://i1qrnm2fgb.execute-api.us-east-1.amazonaws.com/PROD/GET/${AppName}` is an endpoint of an API deployed on AWS API Gateway.
 - This URL is broken down into several components:
 - `https://`: The protocol used for the network request, which is HTTPS (Hypertext Transfer Protocol Secure).
 - `i1qrnm2fgb.execute-api.us-east-1.amazonaws.com`: The domain name, specifically the endpoint for an API hosted on AWS API Gateway.
 - `i1qrnm2fgb`: A unique identifier for the API Gateway endpoint.
 - `us-east-1`: The AWS region where the API Gateway is deployed.
 - `execute-api`: A subdomain for AWS API Gateway.
 - `.amazonaws.com`: The domain for AWS services.
 - `/PROD/GET/${AppName}`: The path of the API request.
 - `/PROD`: The stage name of the API, which typically represents the deployment stage (e.g., "PROD" for production).
 - `/GET`: The resource or operation name defined in the API, which could map to a specific AWS Lambda function or another backend service.
 - `${AppName}`: A dynamic segment in the URL that represents a variable. This value is injected into the URL using JavaScript template literals. It is the value of the `AppName` variable, which is likely the name of an appliance or another entity.
- **Template Literal:**
 - The URL is enclosed in backticks (```) and uses `${AppName}` to dynamically insert the value of the `AppName` variable into the string. This is a feature of JavaScript template literals,

which allow embedding expressions inside a string.

3. `method: 'GET'`

- **HTTP Method:**

- The `method` option specifies the HTTP method for the request. In this case, it is `'GET'`.
- The `'GET'` method is used to request data from a specified resource. It does not modify data on the server; instead, it retrieves data.

4. `headers: { 'Content-Type': 'application/json' }`

- **Headers Object:**

- The `headers` option is an object that specifies additional information to be sent with the HTTP request.
- `'Content-Type': 'application/json'`:
 - The `Content-Type` header indicates the media type of the resource being sent. Since it is a `GET` request, the `Content-Type` header is not required by default, but it can be included to specify that the client expects JSON data in the response.
 - `application/json` specifies that the content type of the expected response is in JSON format.

What This Code Does

- The code sends an asynchronous `GET` request to the specified API Gateway endpoint.
- The URL dynamically includes the value of the `AppName` variable.
- The request uses the `GET` method, which is used to retrieve data from the server.
- The `headers` object specifies that the client expects JSON content in the response.
- The `fetch` function waits for the request to complete, and the result is stored in the `response` variable.

Typical Use Case

- This code is commonly used in web applications to communicate with a backend API.
- In your case, it is likely fetching information related to an appliance from a DynamoDB table or an AWS Lambda function, which is invoked by the API Gateway.

Would you like more details on any specific part of this code or further explanation on how to handle the response?

Two Ways to sent Data

1. PathParameter or QueryStringParameter
2. Using Body

```

const response = await fetch(`https://i1qrnm2fgb.execute-api.us-east-1.amazonaws.com/prod/items?name=${name}`)
  .method: 'GET',
  headers: {
    'Content-Type': 'application/json',
  }
);

// This type is PathParams where we insert a variable value
// in the url

//Two Ways of sending Data

const response = await fetch(`https://i1qrnm2fgb.execute-api.us-east-1.amazonaws.com/prod/items`)
  .method: 'POST',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify(data)
);

// This is using body, where we send the data as json object

```

The two `fetch` requests you've provided are similar in structure, but they differ in several important ways: the HTTP method used, the purpose of the request, and the data being sent or received. Let's break down these differences in detail.

Differences Between the Two `fetch` Requests

1. HTTP Method:

- **First Request:** `method: 'POST'`
 - The first request uses the `POST` method, which is typically used to send data to a server to create or update a resource.
 - `POST` requests often have a body containing data that the server will process (like form data, JSON objects, etc.).
- **Second Request:** `method: 'GET'`
 - The second request uses the `GET` method, which is used to retrieve data from a server.
 - `GET` requests do not have a body; they only fetch information from the server based on the URL and query parameters.

2. URL Structure:

- **First Request URL:**

```
`https://i1qrnm2fgb.execute-api.us-east-1.amazonaws.com/PROD/POST`
```

- This URL points to the `/PROD/POST` path of the API Gateway endpoint. It suggests that this endpoint is meant to handle a `POST` request, likely for creating or updating data.
- The endpoint does not have any additional parameters in the URL, meaning any required data would be sent in the request body.

- **Second Request URL:**

```
`https://i1qrnm2fgb.execute-api.us-east-1.amazonaws.com/PROD/GET/${AppName}`
```

- This URL points to the `/PROD/GET/{AppName}` path of the API Gateway endpoint. The `GET` endpoint is dynamic and includes a path parameter (`${AppName}`), which is substituted by a specific appliance name or value.
- This dynamic URL structure indicates that the request is meant to fetch data related to a particular `AppName`.

3. Purpose of the Request:

- **First Request (`POST`):**

- This request is likely intended to create a new resource or update an existing resource on the server. For example, it could be adding a new appliance to the database or updating its status.
- The server expects data to be sent in the request body, such as a JSON object containing details about the new or updated resource.

- **Second Request (`GET`):**

- This request is used to retrieve information from the server about a specific resource identified by `AppName` .
- It does not modify any data on the server. Instead, it queries the server for information about an existing item.

4. Request Body:

- **First Request (`POST`):**

- `POST` requests usually include a body that contains data being sent to the server. In this example, a body isn't specified in the provided code snippet, but typically it would look like this:

```
body: JSON.stringify({ key: 'value' })
```

- The `body` contains the JSON data that the server will use to create or update a resource.
- **Second Request (`GET`):**
 - `GET` requests do not include a body. The server retrieves data based on the URL parameters or query string provided in the URL.

5. Data Flow:

- **First Request (`POST`):**
 - Data flows from the client to the server. The client sends data that the server processes (e.g., creating or updating a resource in a database).
- **Second Request (`GET`):**
 - Data flows from the server to the client. The server returns data based on the specific `AppName` provided in the URL path.

6. Headers:

- Both requests include the following header:

```
headers: {
  'Content-Type': 'application/json'
}
```

- **First Request (`POST`):**
 - The `Content-Type` header is necessary to tell the server that the request body contains JSON data, which the server should parse accordingly.
- **Second Request (`GET`):**
 - The `Content-Type` header is less important for `GET` requests since there is no body. However, it may indicate that the client expects the response in JSON format.

Summary of Key Differences

Aspect	<code>POST</code> Request	<code>GET</code> Request
HTTP Method	<code>POST</code>	<code>GET</code>
Purpose	Create or update data on the server	Retrieve data from the server
URL Structure	Static (<code>/PROD/POST</code>)	Dynamic (<code>/PROD/GET/\${AppName}</code>)
Request Body	Typically includes a body with data to send	No body; uses path or query parameters for data retrieval
Data Flow	Client to Server	Server to Client
Headers	Indicates content type for data being sent	Indicates expected response format (optional for <code>GET</code>)

Use Case Scenarios

- **POST Request:** Use this when you need to send data to the server to create or update resources (e.g., adding a new appliance or updating its settings).
- **GET Request:** Use this when you want to retrieve data from the server, like fetching the data of an existing appliance using its name.

Would you like more details or examples on how to handle these requests?

How to read This data

To handle these two types of data (**POST** and **GET** requests) in a Lambda function, you need to differentiate between the request methods and read the data accordingly.

Here's how to handle both **POST** and **GET** requests in a Lambda function:

1. Handling a **POST** Request

For a **POST** request, data is usually sent in the request body in JSON format. In your Lambda function, you need to:

- **Parse the body** to extract the data.
- **Process** the data according to your application's requirements.

Example Lambda Function to Handle a **POST** Request:

```
import json
import boto3
from botocore.exceptions import ClientError

# Initialize a DynamoDB client
dynamodb = boto3.resource('dynamodb')
table_name = 'appliance' # Replace with your DynamoDB table name
table = dynamodb.Table(table_name)

def lambda_handler(event, context):
    # Check if the HTTP method is 'POST'
    if event['httpMethod'] == 'POST':
        try:
            # Parse the JSON body from the request
            body = json.loads(event['body'])

            # Extract data from the parsed body
            app_name = body.get('AppName')
            app_status = body.get('Status')

            # Example: Inserting the data into a DynamoDB table
            response = table.put_item(
                Item={
                    'AppName': app_name,
```

```

        'Status': app_status
    }
)

return {
    'statusCode': 201, # HTTP Status Code for 'Created'
    'headers': {
        'Access-Control-Allow-Origin': '*',
        'Access-Control-Allow-Methods': 'POST, OPTIONS',
        'Access-Control-Allow-Headers': 'Content-Type'
    },
    'body': json.dumps({'message': 'Item created successfully'})
}

except ClientError as e:
    # Handle DynamoDB client errors
    return {
        'statusCode': 500,
        'headers': {
            'Access-Control-Allow-Origin': '*',
            'Access-Control-Allow-Methods': 'POST, OPTIONS',
            'Access-Control-Allow-Headers': 'Content-Type'
        },
        'body': json.dumps({'error': str(e)})
    }

# If the HTTP method is not 'POST', handle accordingly
return {
    'statusCode': 400,
    'headers': {
        'Access-Control-Allow-Origin': '*',
        'Access-Control-Allow-Methods': 'POST, OPTIONS',
        'Access-Control-Allow-Headers': 'Content-Type'
    },
    'body': json.dumps({'error': 'Unsupported HTTP method'})
}

```

2. Handling a **GET** Request

For a **GET** request, the data is usually sent as **path parameters** or **query string parameters**. In your Lambda function, you need to:

- **Extract the parameters** from the `event` object.
- **Use the extracted parameters** to perform the necessary operations, like fetching data from database.

Example Lambda Function to Handle a `GET` Request:

```
import json
import boto3
from botocore.exceptions import ClientError

# Initialize a DynamoDB client
dynamodb = boto3.resource('dynamodb')
table_name = 'appliance' # Replace with your DynamoDB table name
table = dynamodb.Table(table_name)

def lambda_handler(event, context):
    # Check if the HTTP method is 'GET'
    if event['httpMethod'] == 'GET':
        try:
            # Extract the AppName from the path parameters
            app_name = event['pathParameters']['AppName']

            # Retrieve the item from DynamoDB
            response = table.get_item(
                Key={
                    'AppName': app_name
                }
            )

            # If item is not found, return a 404 error
            if 'Item' not in response:
                return {
                    'statusCode': 404,
                    'headers': {
                        'Access-Control-Allow-Origin': '*',
                        'Access-Control-Allow-Methods': 'GET, OPTIONS',
                        'Access-Control-Allow-Headers': 'Content-Type'
                    },
                    'body': json.dumps({'message': 'Item not found'})
                }

            # Return the retrieved item
            return {
                'statusCode': 200,
                'headers': {
                    'Access-Control-Allow-Origin': '*',
                    'Access-Control-Allow-Methods': 'GET, OPTIONS',
                    'Access-Control-Allow-Headers': 'Content-Type'
                },
                'body': json.dumps(response['Item'])
            }
        except ClientError as e:
            return {
                'statusCode': 500,
                'body': str(e)
            }
    else:
        return {
            'statusCode': 405,
            'body': 'Method Not Allowed'
        }

```

```

        }

    except ClientError as e:
        # Handle DynamoDB client errors
        return {
            'statusCode': 500,
            'headers': {
                'Access-Control-Allow-Origin': '*',
                'Access-Control-Allow-Methods': 'GET, OPTIONS',
                'Access-Control-Allow-Headers': 'Content-Type'
            },
            'body': json.dumps({'error': str(e)})
        }

    # If the HTTP method is not 'GET', handle accordingly
    return {
        'statusCode': 400,
        'headers': {
            'Access-Control-Allow-Origin': '*',
            'Access-Control-Allow-Methods': 'GET, OPTIONS',
            'Access-Control-Allow-Headers': 'Content-Type'
        },
        'body': json.dumps({'error': 'Unsupported HTTP method'})
    }

```

Key Points for Both Requests

1. Identify the HTTP Method:

- Use the `event['httpMethod']` to determine the type of HTTP request (`GET`, `POST`, etc.).

2. Read Data:

- For `POST` requests, read the body using `event['body']` and parse it as JSON.
- For `GET` requests, extract path parameters or query string parameters from `event['pathParameters']` or `event['queryStringParameters']`.

3. Perform Operations:

- Based on the request type, perform the appropriate action (e.g., inserting data into DynamoDB for `POST` or fetching data for `GET`).

4. Return a Response:

- Return a properly formatted response with the appropriate HTTP status code and headers ensuring CORS headers are set correctly.

Summary

- **POST Request:** Read data from `event['body']`, process it, and return a response.

- **GET Request:** Read data from `event['pathParameters']` or `event['queryStringParameters']`, retrieve the data, and return it.

Would you like more details or examples on specific parts of the Lambda function?

Can I able to send Body to a GET Request.

Technically, you **can** send a body with a `GET` request, but it's not recommended or standard practice. Here's why:

Key Points:

1. **HTTP Specification:** According to the HTTP/1.1 specification, a `GET` request is supposed to retrieve data from a server. The specification does not define what should happen if a `GET` request has a body. Most servers, frameworks, and libraries are designed to ignore the body in a `GET` request.
2. **Browser and Server Behavior:**
 - Browsers like Chrome, Firefox, and others will ignore any body content in a `GET` request.
 - Many server-side libraries and frameworks are not configured to process a body with a `GET` request. They may ignore it, or in some cases, generate an error.
3. **Idempotency and Caching:**
 - `GET` requests are considered **idempotent** (they do not change the server's state) and **cacheable**. Using a body with a `GET` request could break these assumptions and lead to unpredictable behavior.

Alternatives to Sending a Body with `GET`

If you need to send data to a server, here are two recommended alternatives:

1. Use Query Parameters in a `GET` Request:

- Send the data as part of the URL using query parameters.
- Example:

```
const response = await fetch(`https://example.com/api/resource?param1=value1&param2=value2`, {
  method: 'GET',
  headers: {
    'Content-Type': 'application/json',
  }
});
```

- This is the most common approach for sending data in a `GET` request.

2. Use a `POST` Request:

- If you need to send a body, switch to a `POST` request, which is designed to carry a request body.
- Example:

```
const response = await fetch(`https://example.com/api/resource`, {  
  method: 'POST',  
  headers: {  
    'Content-Type': 'application/json',  
  },  
  body: JSON.stringify({ param1: 'value1', param2: 'value2' })  
});
```

Conclusion

While you technically **can** send a body with a `GET` request, it is not standard practice and may not work as expected across different servers and clients. The better approach is to use query parameters or switch to a `POST` request.

Can I send data both as PathParameters and Body in any requests.

Yes, you can send data using both path parameters and the body in a `POST` request. Here's how each is used and how you can combine them:

1. Path Parameters

Path parameters are part of the URL path and are typically used to specify the resource or endpoint. For example:

```
<https://example.com/api/resource/{id}>
```

In this case, `{id}` is a path parameter. You use it to identify a specific resource.

2. Request Body

The request body is used to send data to the server, such as creating or updating a resource. The data is usually in JSON format but can be in other formats like XML or form data.

Combining Path Parameters and Body in a `POST` Request

You can include path parameters in the URL and send data in the body of the request. Here's an example:

Example

Let's say you want to create or update a user profile and use an ID in the path and user details in the body.

- **Endpoint URL:** `https://example.com/api/users/{id}`
- **Path Parameter:** `id` (e.g., `1234`)
- **Request Body:**

```
{  
    "name": "John Doe",  
    "email": "john.doe@example.com",  
    "age": 30  
}
```

JavaScript Fetch Example:

```
const userId = '1234'; // Path parameter  
const userData = {  
    name: 'John Doe',  
    email: 'john.doe@example.com',  
    age: 30  
};  
  
const response = await fetch(`https://example.com/api/users/${userId}`, {  
    method: 'POST',  
    headers: {  
        'Content-Type': 'application/json',  
    },  
    body: JSON.stringify(userData)  
});  
  
const result = await response.json();  
console.log(result);
```

Server-Side Handling

On the server side, you would handle both path parameters and the body:

Example in AWS Lambda with Python:

```
import json  
  
def lambda_handler(event, context):  
    # Extract the path parameter  
    user_id = event['pathParameters']['id']  
  
    # Extract the body data  
    body = json.loads(event['body'])
```

```

name = body.get('name')
email = body.get('email')
age = body.get('age')

# Process the data
response = {
    'statusCode': 200,
    'body': json.dumps({
        'message': f'User {user_id} updated with name {name}, email {email}, age {age}'
    })
}

return response

```

Summary

- Path Parameters:** Used in the URL to specify a resource or identifier.
- Request Body:** Used to send data (e.g., JSON) to the server.

Using both allows you to specify which resource you're interacting with via path parameters and provide detailed data about that resource in the body.

Best way to send data

Here's a table summarizing how you can send data with different HTTP methods (`GET`, `PUT`, `POST`, and `DELETE`), including where you typically place the data:

HTTP Method	Data Location	Description	Example
<code>GET</code>	Query Parameters in URL	Used to retrieve data from the server. Data is typically sent as query parameters.	<code>https://example.com/api/resource?id=1234&name=John</code>
<code>PUT</code>	Path Parameters in URL, Request Body	Used to update or create a resource at a specific URL. Data is sent in the request body.	URL: <code>https://example.com/api/resource/1234</code> Body <code>{"name": "John", "email": "john@example.com"}</code>
<code>POST</code>	Path Parameters in	Used to create a new resource or submit data. Data	URL: <code>https://example.com/api/resource</code> Body <code>{"name": "John", "email": "john@example.com"}</code>

	URL, Request Body	is sent in the request body.	
DELETE	Path Parameters in URL	Used to delete a resource. Data (such as an ID) is typically sent as a path parameter.	URL: https://example.com/api/resource/1234

Detailed Breakdown

1. GET

- **Data Location:** Query Parameters in URL
- **Example:**

```
const response = await fetch('https://example.com/api/resource?id=1234&name=John', {
  method: 'GET',
  headers: {
    'Content-Type': 'application/json',
  },
});
```

2. PUT

- **Data Location:** Path Parameters in URL, Request Body
- **Example:**

```
const response = await fetch('https://example.com/api/resource/1234', {
  method: 'PUT',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    name: 'John',
    email: 'john@example.com'
  })
});
```

3. POST

- **Data Location:** Path Parameters in URL, Request Body
- **Example:**

```
const response = await fetch('https://example.com/api/resource', {
  method: 'POST',
  headers: {
```

```

        'Content-Type': 'application/json',
    },
    body: JSON.stringify({
        name: 'John',
        email: 'john@example.com'
    })
);

```

4. DELETE

- **Data Location:** Path Parameters in URL
- **Example:**

```

const response = await fetch('<https://example.com/api/resource/123>',
    {
        method: 'DELETE',
        headers: {
            'Content-Type': 'application/json',
        }
    });

```

Summary

- **GET:** Data is sent as query parameters in the URL.
- **PUT:** Data is sent in the request body, with path parameters in the URL.
- **POST:** Data is sent in the request body, with optional path parameters in the URL.
- **DELETE:** Data (such as an ID) is sent as a path parameter in the URL.

This table and examples should help you understand how to send data with each HTTP method effectively.

What is Query String Parameters and how to use them.

Using Query String Parameters with `GET` Requests

1. Sending Query String Parameters in JavaScript

To send query string parameters with a `GET` request, you append them to the URL. Here's how you can modify your `fetch` request to use query string parameters:

JavaScript Fetch Example with Query String Parameters:

```

const AppName = 'exampleAppName'; // Replace with actual value
const status = 'active'; // Another query parameter

const response = await fetch(`https://i1qrnm2fgb.execute-api.us-east-1.amazonaws.com/PROD/GET?AppName=${encodeURIComponent(AppName)}&status=${encodeURIComponent(status)}`, {
    method: 'GET',
    headers: {
        'Content-Type': 'application/json',
    }
});

const result = await response.json();
console.log(result);

```

In this example:

- `AppName` and `status` are query parameters.
- `encodeURIComponent` is used to encode the parameters to ensure they are correctly formatted in the URL.

2. Reading Query String Parameters in AWS Lambda

To read query string parameters in AWS Lambda, you access them from the `queryStringParameters` field of the `event` object.

AWS Lambda Example in Python:

```

import json

def lambda_handler(event, context):
    # Extract query string parameters
    app_name = event['queryStringParameters'].get('AppName')
    status = event['queryStringParameters'].get('status')

    # Process the data
    response = {
        'statusCode': 200,
        'body': json.dumps({
            'message': f'AppName: {app_name}, Status: {status}'
        })
    }

    return response

```

In this example:

- `event['queryStringParameters']` contains the query string parameters sent with the request.

- `.get('AppName')` and `.get('status')` are used to safely retrieve the values of the parameters.

Summary

- **JavaScript Fetch Request with Query String Parameters:**
 - Append parameters to the URL after a `?`.
 - Use `encodeURIComponent` to handle special characters in parameter values.
- **AWS Lambda Handling Query String Parameters:**
 - Access query parameters from `event['queryStringParameters']`.
 - Use `.get()` method to retrieve the values.

This approach allows you to send and read data in a `GET` request using query string parameters which can be useful for filtering, sorting, or providing additional context in your requests.

Creating API'S According to the Way we send data

API Gateway Resource Design for Different HTTP Methods

When designing APIs in AWS API Gateway, the way you structure your resources and methods affects how you interact with the API. Here's a breakdown of how to handle different scenarios:

1. Without Path Parameters (Flat Resources)

Example URLs:

- `https://i1qrnm2fgb.execute-api.us-east-1.amazonaws.com/PROD/POST`
- `https://i1qrnm2fgb.execute-api.us-east-1.amazonaws.com/PROD/PUT`

Explanation:

- **Flat Resources:** These URLs represent flat endpoints, meaning they don't include variables dynamic segments in the path.
- **POST/PUT Requests:** These requests usually involve sending data in the body of the request rather than using path parameters to specify resources.

When to Use:

- **POST:** For creating new resources or submitting data. The resource itself is not identified in the URL path.
- **PUT:** For updating or replacing a resource. Typically used when the resource ID is not part of the path, but you still need to identify which resource to update via the request body.

API Gateway Setup:

1. **Create a Resource:** Define the resource path without dynamic segments (e.g., `/POST` or `/PUT`)

2. **Create Methods:** Attach HTTP methods (POST, PUT) to this resource.

2. With Path Parameters (Dynamic Resources)

Example URL:

- `https://i1qrnm2fgb.execute-api.us-east-1.amazonaws.com/PROD/GET/{AppName}`

Explanation:

- **Dynamic Resource:** The URL includes a placeholder `{AppName}`, which is replaced with actual values in requests.
- **Path Parameters:** Used to identify specific resources or elements in the path.

When to Use:

- **GET:** When you need to retrieve a specific resource identified by a unique parameter (e.g., `AppName`).
- **DELETE:** When you need to delete a resource identified by a unique parameter.

API Gateway Setup:

1. **Create a Resource:** Define the resource path with a placeholder for the path parameter (e.g. `/GET/{AppName}`).
2. **Create Methods:** Attach HTTP methods (GET, DELETE) to this resource and configure the integration with Lambda.

3. With Query String Parameters

Example URL:

- `https://i1qrnm2fgb.execute-api.us-east-1.amazonaws.com/PROD/GET?AppName=exampleAppName&status=active`

Explanation:

- **Query String Parameters:** Parameters appended to the end of the URL after a `?`. Used to filter or modify the request.
- **GET Requests:** Commonly used with query parameters to retrieve data based on filtering criteria.

When to Use:

- **GET:** When you need to pass additional data to refine the response, such as search filters, sorting options, or pagination.

API Gateway Setup:

1. **Create a Resource:** Define the base resource path (e.g., `/GET`).
2. **Create Methods:** Attach the GET method to this resource.
3. **Configure Integration:** Ensure that the Lambda function or backend service is set up to handle query string parameters. API Gateway passes these parameters to the Lambda function in the `queryStringParameters` field.

Summary Table

Scenario	Resource URL	Use Case	API Gateway Setup
Without Path Parameters	<code>/POST</code> , <code>/PUT</code>	Creating or updating resources without ID in path	Flat resources; attach methods to the resource
With Path Parameters	<code>/GET/{AppName}</code>	Identifying specific resources (e.g., retrieving by ID)	Dynamic resources define placeholder in the resource path
With Query String Parameters	<code>/GET</code> with query strings (e.g., <code>/GET?AppName=exampleApp&status=active</code>)	Filtering or modifying responses (e.g., search queries)	Base resource path methods handle query strings in request

Best Practices

- **Use Path Parameters** when identifying specific resources by ID or unique identifier.
- **Use Query String Parameters** for filtering, searching, or adding optional parameters to requests.
- **Flat Resources** are suitable for operations where the resource is not identified through the URL path but through the request body.

This approach helps you structure your API endpoints in a way that makes them more intuitive and flexible for different types of requests.

Creating API For Handling Query String Parameters

To set up an API in AWS API Gateway that uses query string parameters, follow these steps:

1. Create or Select an API

1. **Log in to AWS Management Console.**
2. **Navigate to API Gateway.**
3. **Create a new API** or select an existing API:
 - For a new API, click "Create API" and choose the appropriate API type (e.g., HTTP API or REST API).

2. Create a Resource

1. **In the API Gateway console**, select your API.
2. **Navigate to Resources** (for REST API) or directly to the route (for HTTP API).
3. **Create a new resource**:
 - For REST API, click "Actions" and then "Create Resource".
 - Enter a Resource Name and Resource Path (e.g., `/GET` for REST API).

- Click "Create Resource".

3. Add a Method to the Resource

1. **Select the newly created resource.**
2. **Add a method:**
 - For REST API, click "Actions" and then "Create Method".
 - Select `GET` from the dropdown list.
 - Click the checkmark to confirm.

4. Configure the Method

1. **Configure the Integration Request:**
 - **For REST API:**
 - Select "Lambda Function" or your backend service.
 - Configure the integration settings as needed.
 - **For HTTP API:**
 - Configure the route to integrate with Lambda or another service.
2. **Set Up Query String Parameters Handling:**
 - **For REST API:**
 - Click "Integration Request".
 - Under "Mapping Templates", you can map query string parameters to your Lambda function's input.
 - **For HTTP API:**
 - AWS handles query parameters automatically and passes them to your Lambda function.

5. Deploy the API

1. **Create a Deployment Stage:**
 - For REST API: Click "Actions" and then "Deploy API".
 - Select an existing stage or create a new one.
 - Click "Deploy".
2. **For HTTP API:**
 - Deployment is automatically handled, but you may need to configure stages and domain if required.

6. Test the API

1. **Use a tool** like Postman or `curl` to test your API:

- Example URL: `https://your-api-id.execute-api.region.amazonaws.com/PROD/GET?AppName=exampleApp&status=active`
- Send a `GET` request and verify that the query string parameters are correctly processed.

7. Lambda Function Setup

Example Lambda Function in Python:

```
import json

def lambda_handler(event, context):
    # Extract query string parameters
    app_name = event['queryStringParameters'].get('AppName')
    status = event['queryStringParameters'].get('status')

    # Process the data
    response = {
        'statusCode': 200,
        'body': json.dumps({
            'message': f'AppName: {app_name}, Status: {status}'
        })
    }

    return response
```

Summary of Steps

1. **Create or select an API** in API Gateway.
2. **Create a new resource** or use an existing one.
3. **Add a method (e.g., GET)** to the resource.
4. **Configure the method** to integrate with Lambda or another backend.
5. **Deploy the API** to a stage.
6. **Test the API** to ensure it correctly processes query string parameters.
7. **Set up Lambda function** or backend to handle and process query string parameters.

By following these steps, you will have an API configured to handle query string parameters, which can be used for filtering or modifying requests.

After the fetch Api is done

```
const data = await response.json();
```

The line `const data = await response.json();` is used in JavaScript to handle the response from a `fetch` request when interacting with APIs. Here's a detailed breakdown of what it does:

Explanation

1. `await response.json();`:
 - `response`: This is the object returned by the `fetch` function. It represents the HTTP response from the server.
 - `response.json()`: This method reads the response stream to completion and parses it as JSON. It returns a promise that resolves with the result of parsing the body text as JSON.
 - `await`: The `await` keyword is used to wait for the promise returned by `response.json()` to resolve. This allows you to handle the parsed JSON data as a JavaScript object.
2. `const data`:
 - This declares a constant variable `data` that will hold the parsed JSON object.

Usage Example

Here's a full example of how this line might be used in a typical `fetch` request:

```
async function fetchData() {  
    try {  
        // Make a GET request to the API  
        const response = await fetch('<https://api.example.com/data>');  
  
        // Check if the request was successful (status code 200-299)  
        if (!response.ok) {  
            throw new Error(`HTTP error! Status: ${response.status}`);  
        }  
  
        // Parse the JSON response  
        const data = await response.json();  
  
        // Use the JSON data  
        console.log(data);  
    } catch (error) {  
        // Handle errors  
        console.error('Error fetching data:', error);  
    }  
}  
  
fetchData();
```

Detailed Breakdown

1. `fetch('https://api.example.com/data')` : Makes a network request to the specified URL.
2. `await fetch(...)` : Waits for the fetch promise to resolve and returns a `Response` object.
3. `response.ok` : A boolean that indicates whether the response status is in the range 200-299. If not, an error is thrown.
4. `await response.json()` : Waits for the response body to be read and parsed as JSON.
5. `const data` : Contains the parsed JSON object, which can be used in your application.
6. **Error Handling:** If there's an issue with the network request or the response is not OK, an error is thrown and caught in the `catch` block.

When to Use

- **Parsing JSON Responses:** When you expect the server response to be in JSON format and want to work with it as a JavaScript object.
- **Handling Asynchronous Data:** When making asynchronous requests, you use `await` to handle the response data in a readable, synchronous-like fashion.

This approach ensures that your application can handle and process JSON responses from APIs efficiently.

After Getting the results form the API (Backend)

1. We can either send data to console.error tab or
2. Put Data to a element.

Why use Lambda Proxy

Lambda Proxy Integration is a feature in AWS API Gateway that allows you to directly map incoming requests to your Lambda function and return the response from the Lambda function back to the client. This integration affects how requests and responses are handled between API Gateway and Lambda. Here's a comparison of what happens when you use Lambda Proxy Integration versus not using it:

1. Lambda Proxy Integration

How It Works:

- **Request Handling:** API Gateway passes the entire HTTP request (including headers, query string parameters, path parameters, and body) to the Lambda function.
- **Response Handling:** The Lambda function must return a response in a specific format, which API Gateway uses to construct the HTTP response.

Lambda Function Response Format:

```
{
    "statusCode": 200,
    "headers": {
        "Content-Type": "application/json"
    },
    "body": JSON.stringify({
        "message": "Success"
    }),
    "isBase64Encoded": false
}
```

Benefits:

- **Flexibility:** You have complete control over the request and response formats. You can directly access all parts of the HTTP request.
- **Ease of Integration:** Simplifies the integration with Lambda as you don't need to configure mapping templates for request and response transformations.

Example Lambda Function (Python):

```
import json

def lambda_handler(event, context):
    # Extract data from the event
    path = event['path']
    query_string_parameters = event['queryStringParameters']
    headers = event['headers']
    body = json.loads(event['body']) if event.get('body') else {}

    # Process the request and prepare the response
    response = {
        'statusCode': 200,
        'headers': {
            'Content-Type': 'application/json'
        },
        'body': json.dumps({
            'message': 'Success',
            'path': path,
            'query': query_string_parameters,
            'headers': headers,
            'body': body
        })
    }

    return response
```

2. Non-Proxy (Custom) Integration

How It Works:

- **Request Handling:** API Gateway maps incoming requests to the Lambda function using mapping templates. You need to define these templates to extract request parameters and body.
- **Response Handling:** API Gateway uses mapping templates to transform the Lambda function's response into an HTTP response.

Mapping Templates:

- **Request Mapping Template:** Used to transform the incoming request into the format expected by the Lambda function.
- **Response Mapping Template:** Used to transform the Lambda function's response into the format expected by the client.

Benefits:

- **Custom Processing:** Allows you to process and transform request and response data using mapping templates.
- **Separation of Concerns:** Separates request and response handling from the Lambda function, making it easier to manage transformations.

Example Lambda Function (Python):

```
import json

def lambda_handler(event, context):
    # Process the incoming request
    # For example, just return a fixed response
    response = {
        'statusCode': 200,
        'body': json.dumps({'message': 'Success'})
    }

    return response
```

Example Request Mapping Template:

```
{
  "path": "$input.params('path')",
  "queryStringParameters": "$input.params().querystring",
  "headers": "$input.params().header",
  "body": "$input.json('$')"
}
```

Example Response Mapping Template:

```
{
  "statusCode": "$input.path('$.statusCode')",
  "body": "$input.path('$.body')"
}
```

Comparison

Feature	Lambda Proxy Integration	Non-Proxy (Custom) Integration
Request Handling	Passes full request to Lambda function	Uses mapping templates to transform request
Response Handling	Lambda function returns HTTP response directly	Uses mapping templates to transform Lambda response
Flexibility	High flexibility, full control over request/response	Custom processing, requires defining mapping templates
Ease of Use	Easier to set up, fewer configurations needed	More complex setup due to mapping template
Use Cases	Direct Lambda integration, simpler use cases	Advanced scenarios where custom transformations are needed

Summary

- **Lambda Proxy Integration** is often used for its simplicity and flexibility, allowing you to handle the entire request and response within the Lambda function itself.
- **Non-Proxy (Custom) Integration** provides more control over the request and response transformations but requires additional configuration with mapping templates.

Choosing between these approaches depends on your specific use case and whether you need the flexibility of handling raw requests and responses in your Lambda function.

Enabling CORS at Each Stage

1. Enable CORS at Api Gateway
2. Integrate CORS Headers in the lambda function return response.
3. include the below headers in lambda return above the body (Change the method according the method your are using)

```
"headers": {
    "Access-Control-Allow-Origin": "*",
    "Access-Control-Allow-Methods": "POST, OPTIONS",
    "Access-Control-Allow-Headers": "Content-Type",
},
```

Inspect window

1. Use Network tabs and Console Tab to inspect the request. and verify with console.log() methods.
2. Every request contains two things.
 1. Preflight 2. Fetch
3. According to the response from the Fetch, in the "Network Tab" identify the problem.

What are these Fetch and Preflight Requests

When calling an API, you might see two types of requests in your browser's developer tools or network logs: **fetch** and **preflight**. Here's an explanation of each:

1. Fetch Request

What It Is:

- A **fetch request** is the actual request made to the server to retrieve or send data. This request is typically made using JavaScript's `fetch` API or other HTTP libraries (e.g., Axios).

Importance:

- This is the core request where data is exchanged between the client (browser) and the server (API).
- It includes the HTTP method (GET, POST, PUT, DELETE), headers, and possibly a request body, depending on the type of request.
- The fetch request contains the main data payload you are interested in processing or interacting with.

Example Fetch Request:

```
fetch('<https://api.example.com/data>', {
  method: 'GET',
  headers: {
    'Content-Type': 'application/json',
  }
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error('Error:', error));
```

2. Preflight Request

What It Is:

- A **preflight request** is an initial request made by the browser to determine if the actual request is allowed by the server. This occurs in the context of Cross-Origin Resource Sharing (CORS).

Importance:

- CORS is a security feature implemented by browsers to restrict web pages from making requests to a different domain than the one that served the web page.
- A preflight request checks if the actual request is permitted by the server by sending an `OPTIONS` request first. The server responds with headers that indicate whether the actual request can proceed.

When It Happens:

- Preflight requests are triggered for certain types of cross-origin requests, particularly those that are not simple requests. For instance, a request that uses methods other than GET or POST, or requests that include custom headers, will trigger a preflight check.

Example Preflight Request:

- An `OPTIONS` request sent by the browser before the actual request, asking the server what methods and headers are allowed.

Typical Preflight Request:

```
OPTIONS /data HTTP/1.1
Host: api.example.com
Access-Control-Request-Method: POST
Access-Control-Request-Headers: Content-Type
Origin: <https://yourwebsite.com>
```

Typical Preflight Response:

```
HTTP/1.1 204 No Content
Access-Control-Allow-Origin: <https://yourwebsite.com>
Access-Control-Allow-Methods: GET, POST, PUT, DELETE
Access-Control-Allow-Headers: Content-Type
```

Summary

- **Fetch Request:** The actual HTTP request sent to the server to get or send data. This is the request you're primarily interested in and is visible when you use `fetch`, Axios, or similar methods in your code.
- **Preflight Request:** An `OPTIONS` request sent by the browser to check if the actual request is allowed under CORS policies. It helps ensure that the server allows the types of requests you're making.

Why It Matters:

- **Fetch Requests** handle your data transactions with the API.
- **Preflight Requests** are important for ensuring that cross-origin requests comply with security policies and are properly handled by the server.

Understanding these requests helps in troubleshooting issues related to cross-origin requests and ensures that your API interactions are secure and compliant with CORS policies.

Sending Data to Lambda from API Formats

To format the JSON data in a way that AWS Lambda will correctly interpret it, you need to ensure it matches the structure expected by the function. The test event should be in a JSON object format, as shown below:

```
{  
  "pathParameters": {  
    "ApplianceName": "Light"  
  }  
}
```

If you are using the AWS Lambda console or a test environment that requires a stringified version of this JSON, you can use:

```
"{\"pathParameters\": {\"ApplianceName\": \"Light\"}}"
```

This string is properly escaped to ensure it is read as a valid JSON string.

Python Codes

GET-CODE

```
import json  
import boto3  
  
dynamodb = boto3.resource('dynamodb')
```

```

def lambda_handler(event, context):
    # Extract appliance name from query parameters
    AppName = event['pathParameters']['AppName']
    print(AppName)

    if not AppName:
        return {
            'statusCode': 400,
            'headers': {
                'Content-Type': 'application/json',
                'Access-Control-Allow-Origin': '*', # CORS header
                'Access-Control-Allow-Methods': 'GET'
            },
            'body': json.dumps({'error': 'Appliance name is required'})
        }

    # Specify the DynamoDB table name
    table = dynamodb.Table('appliance')

    try:
        # Fetch the item from DynamoDB
        response = table.get_item(
            Key={
                'AppName': AppName
            }
        )
        # print(response)
        item = response.get('Item', None)
        print(item)

        if item != None:
            return {
                'statusCode': 200,
                'headers': {
                    'Content-Type': 'application/json',
                    'Access-Control-Allow-Origin': '*', # CORS header
                    'Access-Control-Allow-Methods': 'GET'
                },
                'body': json.dumps(item)
            }
        else:
            print(response)
            return {
                'statusCode': 404,
                'headers': {
                    'Content-Type': 'application/json',

```

```

        'Access-Control-Allow-Origin': '*', # CORS header
        'Access-Control-Allow-Methods': 'GET'
    },
    'body': json.dumps({'error': 'Item not found'})
}
except Exception as e:
    return {
        'statusCode': 500,
        'headers': {
            'Content-Type': 'application/json',
            'Access-Control-Allow-Origin': '*', # CORS header
            'Access-Control-Allow-Methods': 'GET'
        },
        'body': json.dumps({'error': str(e)})
}

```

POST-CODE

```

import json
import boto3

# Initialize DynamoDB resource
dynamodb = boto3.resource('dynamodb')

def lambda_handler(event, context):
    # Extract AppName and Status from the request body
    body = json.loads(event.get('body', '{}'))
    print(body)
    app_name = body.get('AppName')
    status = body.get('Status')

    # Validation: Check if both AppName and Status are provided
    if not app_name or not status:
        return {
            'statusCode': 400,
            "headers": {
                "Access-Control-Allow-Origin": "*",
                "Access-Control-Allow-Methods": "POST, OPTIONS",
                "Access-Control-Allow-Headers": "Content-Type"},
            'body': json.dumps({'error': 'Both AppName and Status are required'})}
    }

    # Specify the DynamoDB table name
    table_name = 'appliance'

```

```

table = dynamodb.Table(table_name)

try:
    # Check if the item with the given AppName exists
    response = table.get_item(
        Key={'AppName': app_name}
    )

    if 'Item' in response:
        # If the item exists, return it
        return {
            'statusCode': 200,
            "headers": {
                "Access-Control-Allow-Origin": "*",
                "Access-Control-Allow-Methods": "POST, OPTIONS",
                "Access-Control-Allow-Headers": "Content-Type"},
            'body': json.dumps({'message': 'Item already exists', 'item': response['Item']})
    else:
        # If the item does not exist, create it
        table.put_item(
            Item={
                'AppName': app_name,
                'Status': status
            }
        )
        return {
            'statusCode': 201,
            "headers": {
                "Access-Control-Allow-Origin": "*",
                "Access-Control-Allow-Methods": "POST, OPTIONS",
                "Access-Control-Allow-Headers": "Content-Type"},
            'body': json.dumps({'message': 'New item created successfully'})}
}

except Exception as e:
    return {
        'statusCode': 500,
        "headers": {
            "Access-Control-Allow-Origin": "*",
            "Access-Control-Allow-Methods": "POST, OPTIONS",
            "Access-Control-Allow-Headers": "Content-Type"},
        'body': json.dumps({'error': str(e)})}
}

```

PUT-CODE

```
# import json
# import boto3

# # Initialize DynamoDB resource
# dynamodb = boto3.resource('dynamodb')

# def lambda_handler(event, context):
#     # Extract AppName and Status from the request body
#     body = json.loads(event.get('body', '{}'))
#     app_name = body.get('AppName')
#     status = body.get('Status')
#     print('AppName',app_name,'Status',status)

#     # Validation: Check if both AppName and Status are provided
#     if not app_name or not status:
#         return {
#             'statusCode': 400,
#             'body': json.dumps({'error': 'Both AppName and Status are required'})
#         }

#     # Specify the DynamoDB table name
#     table_name = 'appliance'
#     table = dynamodb.Table(table_name)
#     print("passed")

#     try:
#         # Check if the item with the given AppName exists
#         response = table.get_item(
#             Key={'AppName': app_name}
#         )

#         if 'Item' not in response:
#             return {
#                 'statusCode': 404,
#                 'body': json.dumps({'error': 'Item with AppName not found'})
#             }

#         # Update the Status attribute
#         table.update_item(
#             Key={'AppName': app_name},
#             UpdateExpression='SET #st = :status',
#             ExpressionAttributeNames={'#st': 'Status'},
#             ExpressionAttributeValues={':status': status}

#     
```

```

#         )

#         return {
#             'statusCode': 200,
#             'body': json.dumps({'message': 'Status updated successfully'})
#         }

#     except Exception as e:
#         return {
#             'statusCode': 500,
#             'body': json.dumps({'error': str(e)})
#         }

import json
import boto3

# Initialize DynamoDB resource
dynamodb = boto3.resource('dynamodb')

def lambda_handler(event, context):
    # Extract AppName and Status from the request body
    body = json.loads(event.get('body', '{}'))
    print(body)
    app_name = body.get('AppName')
    status = body.get('Status')

    # Validation: Check if both AppName and Status are provided
    if not app_name or not status:
        return {
            'statusCode': 400,
            "headers": {
                "Access-Control-Allow-Origin": "*",
                "Access-Control-Allow-Methods": "POST, OPTIONS",
                "Access-Control-Allow-Headers": "Content-Type",
            },
            'body': json.dumps({'error': 'Both AppName and Status are required'})
        }

    # Specify the DynamoDB table name
    table_name = 'appliance'
    table = dynamodb.Table(table_name)

    try:
        # Check if the item with the given AppName exists

```

```

        response = table.get_item(
            Key={'AppName': app_name}
        )

        if 'Item' not in response:
            return {
                'statusCode': 404,
                "headers": {
                    "Access-Control-Allow-Origin": "*",
                    "Access-Control-Allow-Methods": "POST, OPTIONS",
                    "Access-Control-Allow-Headers": "Content-Type",},
                'body': json.dumps({'error': 'Item with AppName not found'})
            }

        # Update the Status attribute
        table.update_item(
            Key={'AppName': app_name},
            UpdateExpression='SET #st = :status',
            ExpressionAttributeNames={'#st': 'Status'},
            ExpressionAttributeValues={':status': status}
        )

        return {
            'statusCode': 200,
            "headers": {
                "Access-Control-Allow-Origin": "*",
                "Access-Control-Allow-Methods": "POST, OPTIONS",
                "Access-Control-Allow-Headers": "Content-Type",
            },
            'body': json.dumps({'message': 'Status updated successfully'})
        }

    except Exception as e:
        return {
            'statusCode': 500,
            "headers": {
                "Access-Control-Allow-Origin": "*",
                "Access-Control-Allow-Methods": "POST, OPTIONS",
                "Access-Control-Allow-Headers": "Content-Type",
            },
            'body': json.dumps({'error': str(e)})
        }

```

DELETE-CODE

```

import json
import boto3
from botocore.exceptions import ClientError

# Initialize a DynamoDB client
dynamodb = boto3.resource('dynamodb')
table_name = 'appliance' # Replace with your DynamoDB table name
table = dynamodb.Table(table_name)

def lambda_handler(event, context):
    # Extract the AppName from the path parameters
    try:
        AppName = event['pathParameters']['AppName']
    except KeyError:
        return {
            'statusCode': 400,
            'body': json.dumps({'message': 'AppName is required in the path parameters'})
    }

    try:
        # Check if the item exists
        response = table.get_item(
            Key={
                'AppName': AppName # Replace 'AppName' with your partition key name
            }
        )
        print("GetItem response:", response)

        # If item is not found, return a 404 error
        if 'Item' not in response:
            return {
                'statusCode': 404,
                "headers": {
                    "Access-Control-Allow-Origin": "*",
                    "Access-Control-Allow-Methods": "DELETE, OPTIONS",
                    "Access-Control-Allow-Headers": "Content-Type",
                },
                'body': json.dumps({'message': 'Item not found'})
            }

        # If the item exists, proceed to delete it
        delete_response = table.delete_item(
            Key={
                'AppName': AppName
            }
    
```

```

        )
        print("DeleteItem response:", delete_response)

    return {
        'statusCode': 200,
        "headers": {
            "Access-Control-Allow-Origin": "*",
            "Access-Control-Allow-Methods": "DELETE, OPTIONS",
            "Access-Control-Allow-Headers": "Content-Type",
        },
        'body': json.dumps({'message': 'Item deleted successfully'})
    }

except ClientError as e:
    # Handle DynamoDB client errors
    print("DynamoDB ClientError:", e.response['Error']['Message'])
    return {
        'statusCode': 500,
        "headers": {
            "Access-Control-Allow-Origin": "*",
            "Access-Control-Allow-Methods": "DELETE, OPTIONS",
            "Access-Control-Allow-Headers": "Content-Type",
        },
        'body': json.dumps({'error': e.response['Error']['Message']})
    }

except Exception as e:
    # Catch any other exceptions and print them
    print("Exception:", str(e))
    return {
        'statusCode': 500,
        "headers": {
            "Access-Control-Allow-Origin": "*",
            "Access-Control-Allow-Methods": "DELETE, OPTIONS",
            "Access-Control-Allow-Headers": "Content-Type",
        },
        'body': json.dumps({'error': 'Internal server error: ' + str(e)})
    }

```