

Single Calculator Microservice Expose HTTPS methods with Lambda Function URL

Overview:

You're going to create a **calculator microservice** using **AWS Lambda** and expose it via a **Lambda Function URL**. This service will accept POST HTTP requests containing numbers and an operator, perform the calculation, and return the result.

Steps Involved:

1. Design the Event JSON Object

- **Goal:** Define the structure of the request data that the client will send to the Lambda function.
- **What's Included:** The JSON object will contain:
 - num1: First number.
 - num2: Second number.
 - operator: The operation to perform (e.g., +, -, *, /).

2. Develop the Lambda Function

- **Goal:** Write the function that will process the input data (numbers and operator), perform the calculation, and return the result.
- **How it Works:** The Lambda function will:
 - Take the numbers and operator from the request.
 - Perform the specified calculation.
 - Return the result back to the client.

3. Expose the Lambda Function via HTTP

- **Goal:** Make your Lambda function accessible via HTTPS, so clients can send HTTP POST requests to it.
- **How it Works:** AWS provides a **Lambda Function URL**, which allows you to expose the function as a web endpoint. You will configure this to handle incoming POST requests.

4. (Optional) Create ZIP File for Lambda

- **Goal:** If your function uses external libraries or dependencies, you'll package the code into a ZIP file before uploading it to AWS Lambda.
- **When to Use:** If you need to include extra files or dependencies that aren't in the default Lambda environment, this step is required.

5. Configure Lambda (Optional)

- **Goal:** Adjust your Lambda function's settings, such as memory, execution time, and permissions, to make sure it's optimized for your use case.

6. Test Using Postman

- **Goal:** Send test requests to your Lambda function to ensure it's working as expected.
- **How to Test:** Use **Postman** to send a **POST** request to the Lambda function URL with the JSON payload, and check that the function performs the calculation correctly.

7. Final Testing and Validation

- **Goal:** Perform final tests with different inputs to ensure the Lambda function is handling various cases correctly (e.g., different operations and edge cases like division by zero).

Summary of Steps:

1. **Design the Input:** Decide what data (numbers and operator) will be sent to the Lambda function.
2. **Write the Lambda Function:** Create a function to process the data and calculate the result.
3. **Expose via HTTP:** Set up a Lambda Function URL to handle HTTP POST requests.
4. **Package Code (If needed):** If required, zip and upload the function with dependencies.
5. **Configure Lambda:** Set Lambda function settings like memory and timeout.
6. **Test with Postman:** Send test requests to ensure the function works.
7. **Final Testing:** Validate the function with different inputs.

To begin with the Lab

1. The first step is to create our lambda function. Open your AWS Console, search for lambda, and click on the create function from its dashboard.
2. Choose the author from scratch option, give your function a name, choose the runtime as node.js, and create your lambda function.

Basic information

Function name

Enter a name that describes the purpose of your function.

Function name must be 1 to 64 characters, must be unique to the Region, and can't include spaces. Valid characters are a-z, A-Z, 0-9, hyphens (-), and underscores (_).

Runtime Info

Choose the language to use to write your function. Note that the console code editor supports only Node.js, Python, and Ruby.



Architecture Info

Choose the instruction set architecture you want for your function code.

- x86_64
- arm64

Permissions Info

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

▶ Change default execution role

3. Once your function is created go inside, and go to the configuration tab in your function.
4. Here you need to navigate to the function URL and click on Create function URL.

The screenshot shows the AWS Lambda Configuration page. At the top, there are tabs: Code, Test, Monitor, Configuration (which is selected and highlighted in blue), Aliases, and Versions. On the left, a sidebar lists General configuration, Triggers, Permissions, Destinations, Environment variables, and Function URL. The 'Function URL' item is highlighted with a red box. The main content area has a header 'Function URL Info' and a 'Create function URL' button. Below it, a message says 'No Function URL' and 'No Function URL is configured.' with another 'Create function URL' button, also highlighted with a red box.

5. Now here you need to choose None for the Authentication type and scroll down to expand the additional features.

Configure Function URL

Function URL Info

Use function URLs to assign HTTP(S) endpoints to your Lambda function.

Auth type

Choose the auth type for your function URL. [Learn more](#)

AWS_IAM

Only authenticated IAM users and roles can make requests to your function URL.

NONE

Lambda won't perform IAM authentication on requests to your function URL. The URL endpoint will be public unless you implement your own authorization logic in your function.

Function URL permissions

i When you choose auth type **NONE**, Lambda automatically creates the following resource-based policy and attaches it to your function. This policy makes your function public to anyone with the function URL. You can edit the policy later. To limit access to authenticated IAM users and roles, choose auth type [AWS_IAM](#).

6. In the additional settings you need to enable the CORS feature and then scroll down and save your function URL.

▼ Additional settings

Invoke mode

Choose how your function returns responses. [Learn more](#)

BUFFERED (default)

The invocation results are available when the payload is complete. Response payload max size: 6 MB

RESPONSE_STREAM

Stream the invocation results. Streaming responses incur additional costs. Refer to the documentation for payload size limitations.

[Learn more](#)

Configure cross-origin resource sharing (CORS)

Use CORS to allow access to your function URL from any domain. You can also use CORS to control access for specific HTTP headers and methods in requests to your function URL. [Learn more](#)

Allow origin

Add the origins that can access your function URL. You can list any number of specific origins. Alternatively, you can grant access to all origins with the wildcard character (*). For example: https://www.example.com, https://*, or *.)

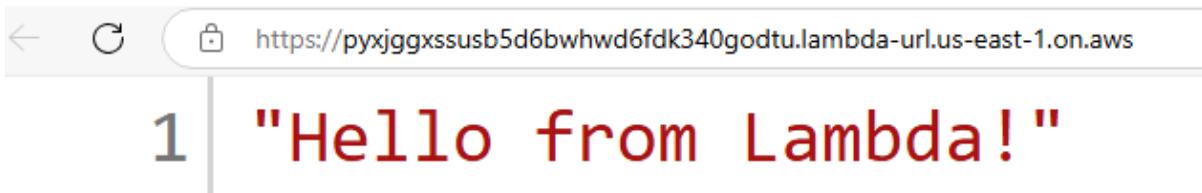
*

[Add new value](#)

7. In the function overview you will see that now you have a Function URL open this in a new tab.

The screenshot shows the 'Function overview' page for a Lambda function named 'function-with-url'. The function has no layers. A red box highlights the 'Function URL' section, which contains the URL <https://pyxjggxssusb5d6bwhwd6fdk340godtu.lambda-url.us-east-1.on.aws/>.

8. So, using the function URL you have directly invoked your lambda function.



9. Now you need to open the code section of your function and add the line that you can see on number 2. This is typically used for **debugging** and **logging** to understand the structure and contents of the event object in AWS Lambda or any other JavaScript application.

```
console.log("EVENT: \n" + JSON.stringify(event, null, 2));
```

The screenshot shows the AWS Lambda code editor for a file named 'index.mjs'. The code defines a handler function that logs the event object to the console. A red box highlights the line of code `console.log("EVENT: \n" + JSON.stringify(event, null, 2));`.

```
JS index.mjs X
JS index.mjs > [(handler]
1  export const handler = async (event) => {
2    console.log("EVENT: \n" + JSON.stringify(event, null, 2));
3    // TODO implement
4    const response = {
5      statusCode: 200,
6      body: JSON.stringify('Hello from Lambda!'),
7    };
8    return response;
9  };
10
```

10. Once you have made the changes deploy them and now once again open the function URL to invoke your function.

11. After invoking your function, you need to open the CloudWatch logs for your function and open the latest log stream.

The screenshot shows the AWS CloudWatch Log Streams interface. At the top, there are tabs for Log streams, Tags, Anomaly detection, Metric filters, Subscription filters, and Contributor Insights. The Log streams tab is selected. Below the tabs, there's a search bar with placeholder text "Filter log streams or try prefix search". To the right of the search bar are buttons for Delete, Create log stream, and Search all log streams. There are also checkboxes for "Exact match" and "Show expired". A status indicator shows "1" log stream. The main table lists two log streams:

<input type="checkbox"/>	Log stream	Last event time
<input type="checkbox"/>	2024/11/15/[\$LATEST]f24d5bba8a96462db9a8c8858bf3fc5	2024-11-15 16:25:45 (UTC+05:30)
<input type="checkbox"/>	2024/11/15/[\$LATEST]bb0d3fff97b94931a17c898c8ef65ed1	2024-11-15 16:18:46 (UTC+05:30)

12. The main purpose of adding that line of code was to get the information about the invocation of our function.

The screenshot shows a single log entry from November 15, 2024, at 16:25:45.890 UTC. The log entry is an incoming event with the following details:

2024-11-15T16:25:45.890+05:30 2024-11-15T10:55:45.890Z f23eac55-4259-4686-aee4-1ff46ba99fb INFO EVENT: { "version": "2.0", "routeKey": "\$default", "rawPath": "/", "rawQueryString": "", "headers": { "sec-fetch-mode": "navigate", "content-length": "0", "x-amzn-tls-version": "TLSv1.3", "sec-fetch-site": "cross-site", "x-forwarded-proto": "https", "accept-language": "en-US,en;q=0.9,en-IN;q=0.8", "x-forwarded-port": "443", "x-forwarded-for": "103.226.202.230", "sec-fetch-user": "1", "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7",

13. Now we are going to change the function code and then invoke our function. So, use the code given below and deploy it in your lambda function.

```

export const handler = async (event) => {
    console.log('Received event:', JSON.stringify(event, null, 2));

    const method = event.requestContext.http.method;
    const queryParam = event.queryStringParameters.message;
    console.log(`Received ${method} request with ${queryParam}`);

    const response = {
        statusCode: 200,
        body: JSON.stringify(`Hello from ${queryParam}`),
    };
    return response;
}

```

```

JS index.mjs  X

JS index.mjs > ...
1  exports.handler = async (event) => {
2    console.log('Received event:', JSON.stringify(event, null, 2));
3
4    // fetch method and querystring
5    const method = event.requestContext.http.method;
6    const queryParam = event.queryStringParameters.message;
7    console.log(`Received ${method} request with ${queryParam}`);
8
9    const response = {
10      statusCode: 200,
11      body: JSON.stringify(`Hello from ${queryParam}`),
12    };
13    return response;
14  }
15

```

14. Now you need to open Postman on your laptop and copy the function URL from your lambda function. Then you need to create a new workspace in your Postman and choose the GET method. After that paste your URL and append it with a message as you can see in the snapshot. You will get a Hello message which means that our function is working fine.

The screenshot shows the Postman interface with the following details:

- URL:** https://pyxjggxssusb5d6bwhwd6fdk340godtu.lambda-url.us-east-1.on.aws?message=cloudfreeks
- Method:** GET
- Params:** message (with value "cloudfreeks")
- Body:** "Hello from cloudfreeks"
- Status:** 200 OK
- Headers:** (6)
- Tests:** (1)
- Body Content:** "Hello from cloudfreeks"

15. Go to the CloudWatch of your lambda function and view your events.
 16. Now we are going to design the lambda function code. This will be our final code for the calculator microservice. Use the below code and deploy it on your lambda function.

```

export const handler = async (event) => {
    console.log('Received event:', JSON.stringify(event, null, 2));

```

```

// Check if event.body exists and is not empty before parsing
if (!event.body) {
    return {
        statusCode: 400,
        body: JSON.stringify({
            error: "Request body is missing or empty"
        })
    };
}

let payload;
try {
    // Try parsing the body
    payload = JSON.parse(event.body);
} catch (error) {
    // If parsing fails, return an error response
    return {
        statusCode: 400,
        body: JSON.stringify({
            error: "Invalid JSON format in request body",
            details: error.message
        })
    };
}

let result = 0;
try {
    if (payload.a === undefined || payload.b === undefined || payload.op === undefined) {
        throw new Error(`event should have properties a, b, and op: ${JSON.stringify(payload)}`);
    }

    switch (payload.op) {
        case "+":
            result = payload.a + payload.b;
            break;
        case "-":
            result = payload.a - payload.b;
            break;
        case "*":
            result = payload.a * payload.b;
            break;
        case "/":
            result = payload.b === 0 ? NaN : payload.a / payload.b;
    }
}

```

```

        break;
    default:
        throw new Error(`Unsupported operation: ${payload.op}`);
    }
    console.log('Result is : ', result);
} catch (error) {
    console.error(error);
    return {
        statusCode: 400,
        body: JSON.stringify({
            error: "Error processing the request",
            details: error.message
        }),
    };
}

return {
    statusCode: 200,
    body: JSON.stringify({
        processed: true,
        result: result
    }),
};
};

```

17. Once the code is deployed open the function URL once more. You should get this type of error shown below.



A screenshot of a web browser window. The address bar shows the URL: <https://pyxjggxssusb5d6bwhwd6fdk340godtu.lambda-url.us-east-1.on.aws>. The page content displays a JSON object with three lines of code:

```

1 {
2     "error": "Request body is missing or empty"
3 }

```

18. Now go to Postman and use the POST method there you need to paste the function URL and then choose **body and from body choose raw then JSON**.
19. In the end put the same JSON code as shown below and click on Send to invoke the request.
20. You will get the same output, based on the operator you choose your calculator will work.

POST https://pyxjggxssusb5

https://pyxjggxssusb5d6bwhwd6fdk340godtu.lambda-url.us-east-1.on.aws

Send

Body (7)

Params Authorization Headers (7) Body (7) Scripts Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```

1 {
2   "a": 2,
3   "b": 5,
4   "op": "+"
5 }
```

Beautify

200 OK 493 ms 331 B

Pretty Raw Preview Visualize JSON

```

1 {
2   "processed": true,
3   "result": 7
4 }
```

21. As you can see below, we changed the operator from addition to multiplication and based on our operator we get the output.

POST https://pyxjggxssusb5

https://pyxjggxssusb5d6bwhwd6fdk340godtu.lambda-url.us-east-1.on.aws

Send

Params Authorization Headers (7) Body (7) Scripts Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```

1 {
2   "a": 2,
3   "b": 5,
4   "op": "*"
5 }
```

Beautify

200 OK 698 ms 332 B

Pretty Raw Preview Visualize JSON

```

1 {
2   "processed": true,
3   "result": 10
4 }
```

22. Once you are done using the calculator delete all the resources. Start with the lambda function and then your CloudWatch logs.