Set up Lambda proxy integrations in API Gateway

PDF | RSS

Topics

Understand API Gateway Lambda proxy integration

Amazon API Gateway Lambda proxy integration is a simple, powerful, and nimble mechanism to build an API with a setup of a single API method. The Lambda proxy integration allows the client to call a single Lambda function in the backend. The function accesses many resources or features of other AWS services, including calling other Lambda functions.

In Lambda proxy integration, when a client submits an API request, API Gateway passes to the integrated Lambda function an event object, except that the order of the request parameters is not preserved. This request data includes the request headers, query string parameters, URL path variables, payload, and API configuration data. The configuration data can include current deployment stage name, stage variables, user identity, or authorization context (if any). The backend

Lambda function parses the incoming request data to determine the response that it returns. For API Gateway to pass the Lambda output as the API response to the client, the Lambda function must return the result in this format.

Because API Gateway doesn't intervene very much between the client and the backend Lambda function for the Lambda proxy integration, the client and the integrated Lambda function can adapt to changes in each other without breaking the existing integration setup of the API. To enable this, the client must follow application protocols enacted by the backend Lambda function.

You can set up a Lambda proxy integration for any API method. But a Lambda proxy integration is more potent when it is configured for an API method involving a generic proxy resource. The generic proxy resource can be denoted by a special templated path variable of `{proxy+}`, the catch-all `ANY` method placeholder, or both. The client can pass the input to the backend Lambda function in the incoming request as request parameters or applicable payload. The request parameters include headers, URL path variables, query string parameters, and the applicable payload. The integrated Lambda function verifies all of the input sources before processing the request and responding to the client with meaningful error messages if any of the required input is missing.

When calling an API method integrated with the generic HTTP method of `ANY` and the generic resource of `{proxy+}`, the client submits a request with a particular HTTP method in place of `ANY`. The client also specifies a particular URL path instead of `{proxy+}`, and includes any required headers, query string parameters, or an applicable payload.

The following list summarizes runtime behaviors of different API methods with the Lambda proxy integration:

- `ANY /{proxy+}`: The client must choose a particular HTTP method, must set a particular resource path hierarchy, and can set any headers, query string parameters, and applicable payload to pass the data as input to the integrated Lambda function.

- `ANY /res`: The client must choose a particular HTTP method and can set any headers, query string parameters, and applicable payload to pass the data as input to the integrated Lambda function.
- `GET|POST|PUT|... /{proxy+}`: The client can set a particular resource path hierarchy, any headers, query string parameters, and applicable payload to pass the data as input to the integrated Lambda function.
- `GET|POST|PUT|... /res/{path}/...`: The client must choose a particular path segment (for the `{path}` variable) and can set any request headers, query string parameters, and applicable payload to pass input data to the integrated Lambda function.
- `GET|POST|PUT|... /res`: The client can choose any request headers, query string parameters, and applicable payload to pass input data to the integrated Lambda function.

Both the proxy resource of `{proxy+}` and the custom resource of `{custom}` are expressed as templated path variables. However `{proxy+}` can refer to any resource along a path hierarchy, while `{custom}` refers to a particular path segment only. For example, a grocery store might organize its online product inventory by department names, produce categories, and product types. The grocery store's website can then represent available products by the following templated path variables of custom resources: `/{department}/{produce-category}/{product-type}`. For example, apples are represented by `/produce/fruit/apple` and carrots by `/produce/vegetables/carrot`. It can also use `/{proxy+}` to represent any department, any produce category, or any product type that a customer can search for while shopping in the online store. For example, `/{proxy+}` can refer to any of the following items:

- `/produce`
- `/produce/fruit`
- `/produce/vegetables/carrot`

To let customers search for any available product, its produce category, and the associated store department, you can expose

a single method of `GET /{proxy+}` with read-only permissions. Similarly, to allow a supervisor to update the `produce` department's inventory, you can set up another single method of `PUT /produce/{proxy+}` with read/write permissions. To allow a cashier to update the running total of a vegetable, you can set up a `POST /produce/vegetables/{proxy+}` method with read/write permissions. To let a store manager perform any possible action on any available product, the online store developer can expose the `ANY /{proxy+}` method with read/write permissions. In any case, at run time, the customer or the employee must select a particular product of a given type in a chosen department, a specific produce category in a chosen department, or a specific department.

For more information about setting up API Gateway proxy integrations, see Set up a proxy integration with a proxy resource.

Proxy integration requires that the client have more detailed knowledge of the backend requirements. Therefore, to ensure optimal app performance and user experience, the backend developer must communicate clearly to the client developer the requirements of the backend, and provide a robust error feedback mechanism when the requirements are not met.

Support for multi-value headers and query string parameters

API Gateway supports multiple headers and query string parameters that have the same name. Multi-value headers as well as single-value headers and parameters can be combined in the same requests and responses. For more information, see Input format of a Lambda function for proxy integration and Output format of a Lambda function for proxy integration.

Set up a proxy resource with Lambda proxy integration

To set up a proxy resource with the Lambda proxy integration type, create an API resource with a greedy path parameter (for example, `/parent/{proxy+}`) and integrate this resource with a Lambda function backend (for example, `arn:aws:lambda:us-west-2:123456789012:function:SimpleLambda4ProxyResource`) on the `ANY` method. The greedy path parameter must be at the end of the API resource path. As with a non-proxy resource, you can set up the proxy resource by using the API Gateway console, importing an OpenAPI definition file, or calling the API Gateway REST API directly.

The following OpenAPI API definition file shows an example of an API with a proxy resource that is integrated with a Lambda function named `SimpleLambda4ProxyResource`.

OpenAPI 3.0 | OpenAPI 2.0

```
{
   "openapi": "3.0.0",
   "info": {
      "version": "2016-09-12T17:50:37Z",
      "title": "ProxyIntegrationWithLambda"
   },
   "paths": {
      "/{proxy+}": {
         "x-amazon-apigateway-any-method": {
            "parameters": [
               {
                  "name": "proxy",
                  "in": "path",
                  "required": true,
                  "schema": {
                     "type": "string"
                  }
               }
            ],
            "responses": {},
            "x-amazon-apigateway-integration":
```

```
                "responses": {
                   "default": {
                      "statusCode": "200"
                   }
                },
                "uri": "arn:aws:apigateway:us-e
                "passthroughBehavior": "when_no
                "httpMethod": "POST",
                "cacheNamespace": "roq9wj",
                "cacheKeyParameters": [
                   "method.request.path.proxy"
                ],
                "type": "aws_proxy"
             }
          }
       }
    },
    "servers": [
       {
          "url": "https://gy415nuibc.execute-ap
          "variables": {
             "basePath": {
               "default": "/testStage"
             }
          }
       }
    ]
}
```

In Lambda proxy integration, at run time, API Gateway maps an incoming request into the input `event` parameter of the Lambda function. The input includes the request method, path, headers, any query string parameters, any payload, associated context, and any defined stage variables. The input format is explained in Input format of a Lambda function for proxy integration. For API Gateway to map the Lambda output to HTTP responses successfully, the Lambda function must output the result in the format described in Output format of a Lambda function for proxy integration.

In Lambda proxy integration of a proxy resource through the `ANY` method, the single backend Lambda function serves as the event handler for all requests through the proxy resource. For example, to log traffic patterns, you can have a mobile device send its location information of state, city, street, and building by submitting a request with `/state/city/street/house` in the URL path for the proxy resource. The backend Lambda function can then parse the URL path and insert the location tuples into a DynamoDB table.

Set up Lambda proxy integration using the AWS CLI

In this section, we show how to use AWS CLI to set up an API with the Lambda proxy integration.

> **Note**
>
> For detailed instructions for using the API Gateway console to configure a proxy resource with the Lambda proxy integration, see Tutorial: Build a Hello World REST API with Lambda proxy integration.

As an example, we use the following sample Lambda function as the backend of the API:

```
export const handler = function(event, context,a
    console.log('Received event:', JSON.stringify(
    var res ={
        "statusCode": 200,
        "headers": {
            "Content-Type": "*/*"
        }
    };
    var greeter = 'World';
    if (event.greeter && event.greeter!=="") {
        greeter =  event.greeter;
    } else if (event.body && event.body !== "") {
        var body = JSON.parse(event.body);
```

```
        if (body.greeter && body.greeter !== "") {
            greeter = body.greeter;
        }
    } else if (event.queryStringParameters && event
        greeter = event.queryStringParameters.greet
    } else if (event.multiValueHeaders && event.mul
        greeter = event.multiValueHeaders.greeter.j
    } else if (event.headers && event.headers.greet
        greeter = event.headers.greeter;
    }

    res.body = "Hello, " + greeter + "!";
    callback(null, res);
};
```

Comparing this to the Lambda custom integration setup, the input to this Lambda function can be expressed in the request parameters and body. You have more latitude to allow the client to pass the same input data. Here, the client can pass the greeter's name in as a query string parameter, a header, or a body property. The function can also support the Lambda custom integration. The API setup is simpler. You do not configure the method response or integration response at all.

To set up a Lambda proxy integration using the AWS CLI

1. Call the `create-rest-api` command to create an API:

   ```
   aws apigateway create-rest-api --name 'Hell
   ```

   Note the resulting API's `id` value (`te6si5ach7`) in the response:

   ```
   {
       "name": "HelloWorldProxy (AWS CLI)",
       "id": "te6si5ach7",
       "createdDate": 1508461860
   }
   ```

You need the API `id` throughout this section.

2. Call the `get-resources` command to get the root resource `id`:

```
aws apigateway get-resources --rest-api-id
```

The successful response is shown as follows:

```
{
    "items": [
        {
            "path": "/",
            "id": "krznpq9xpg"
        }
    ]
}
```

Note the root resource `id` value (`krznpq9xpg`). You need it in the next step and later.

3. Call `create-resource` to create an API Gateway Resource of `/greeting`:

```
aws apigateway create-resource --rest-api-id
      --region us-west-2 \
      --parent-id krznpq9xpg \
      --path-part {proxy+}
```

The successful response is similar to the following:

```
{
    "path": "/{proxy+}",
    "pathPart": "{proxy+}",
    "id": "2jf6xt",
    "parentId": "krznpq9xpg"
}
```

Note the resulting `{proxy+}` resource's `id` value

(`2jf6xt`). You need it to create a method on the `/{proxy+}` resource in the next step.

4. Call `put-method` to create an `ANY` method request of `ANY /{proxy+}`:

```
aws apigateway put-method --rest-api-id te6
       --region us-west-2 \
       --resource-id 2jf6xt \
       --http-method ANY \
       --authorization-type "NONE"
```

The successful response is similar to the following:

```
{
    "apiKeyRequired": false,
    "httpMethod": "ANY",
    "authorizationType": "NONE"
}
```

This API method allows the client to receive or send greetings from the Lambda function at the backend.

5. Call `put-integration` to set up the integration of the `ANY /{proxy+}` method with a Lambda function, named `HelloWorld`. This function responds to the request with a message of `"Hello, {name}!"`, if the `greeter` parameter is provided, or `"Hello, World!"`, if the query string parameter is not set.

```
aws apigateway put-integration \
        --region us-west-2 \
        --rest-api-id te6si5ach7 \
        --resource-id 2jf6xt \
        --http-method ANY \
        --type AWS_PROXY \
        --integration-http-method POST \
        --uri arn:aws:apigateway:us-west-2:lamb
        --credentials arn:aws:iam::123456789012
```

> **Important**
>
> For Lambda integrations, you must use the HTTP method of `POST` for the integration request, according to the specification of the Lambda service action for function invocations. The IAM role of `apigAwsProxyRole` must have policies allowing the `apigateway` service to invoke Lambda functions. For more information about IAM permissions, see API Gateway permissions model for invoking an API.

The successful output is similar to the following:

```
{
    "passthroughBehavior": "WHEN_NO_MATCH",
    "cacheKeyParameters": [],
    "uri": "arn:aws:apigateway:us-west-2:lambda
    "httpMethod": "POST",
    "cacheNamespace": "vvom7n",
    "credentials": "arn:aws:iam::1234567890:rol
    "type": "AWS_PROXY"
}
```

Instead of supplying an IAM role for `credentials`, you can call the add-permission command to add resource-based permissions. This is what the API Gateway console does.

6. Call `create-deployment` to deploy the API to a `test` stage:

```
aws apigateway create-deployment --rest-api
```

7. Test the API using the following cURL commands in a terminal.

   Calling the API with the query string parameter of `?greeter=jane`:

```
curl -X GET 'https://te6si5ach7.execute-api
```

Calling the API with a header parameter of `greeter:jane`:

```
curl -X GET https://te6si5ach7.execute-api.
  -H 'content-type: application/json' \
  -H 'greeter: jane'
```

Calling the API with a body of `{"greeter":"jane"}`:

```
curl -X POST https://te6si5ach7.execute-api
  -H 'content-type: application/json' \
  -d '{ "greeter": "jane" }'
```

In all the cases, the output is a 200 response with the following response body:

```
Hello, jane!
```

Input format of a Lambda function for proxy integration

In Lambda proxy integration, API Gateway maps the entire client request to the input `event` parameter of the backend Lambda function. The following example shows the structure of an event that API Gateway sends to a Lambda proxy integration.

```
{
  "resource": "/my/path",
  "path": "/my/path",
  "httpMethod": "GET",
  "headers": {
    "header1": "value1",
    "header2": "value1,value2"
```

```
  },
  "multiValueHeaders": {
    "header1": [
      "value1"
    ],
    "header2": [
      "value1",
      "value2"
    ]
  },
  "queryStringParameters": {
    "parameter1": "value1,value2",
    "parameter2": "value"
  },
  "multiValueQueryStringParameters": {
    "parameter1": [
      "value1",
      "value2"
    ],
    "parameter2": [
      "value"
    ]
  },
  "requestContext": {
    "accountId": "123456789012",
    "apiId": "id",
    "authorizer": {
      "claims": null,
      "scopes": null
    },
    "domainName": "id.execute-api.us-east-1.amazon
    "domainPrefix": "id",
    "extendedRequestId": "request-id",
    "httpMethod": "GET",
    "identity": {
      "accessKey": null,
      "accountId": null,
      "caller": null,
      "cognitoAuthenticationProvider": null,
      "cognitoAuthenticationType": null,
      "cognitoIdentityId": null,
      "cognitoIdentityPoolId": null,
```

```
      "principalOrgId": null,
      "sourceIp": "IP",
      "user": null,
      "userAgent": "user-agent",
      "userArn": null,
      "clientCert": {
        "clientCertPem": "CERT_CONTENT",
        "subjectDN": "www.example.com",
        "issuerDN": "Example issuer",
        "serialNumber": "a1:a1:a1:a1:a1:a1:a1:a1:a
        "validity": {
          "notBefore": "May 28 12:30:02 2019 GMT",
          "notAfter": "Aug  5 09:36:04 2021 GMT"
        }
      }
    },
    "path": "/my/path",
    "protocol": "HTTP/1.1",
    "requestId": "id=",
    "requestTime": "04/Mar/2020:19:15:17 +0000",
    "requestTimeEpoch": 1583349317135,
    "resourceId": null,
    "resourcePath": "/my/path",
    "stage": "$default"
  },
  "pathParameters": null,
  "stageVariables": null,
  "body": "Hello from Lambda!",
  "isBase64Encoded": false
}
```


Note

In the input:

- The `headers` key can only contain single-value headers.
- The `multiValueHeaders` key can contain multi-value headers as well as single-value headers.
- If you specify values for both `headers` and `multiValueHeaders`, API Gateway merges them

into a single list. If the same key-value pair is specified in both, only the values from `multiValueHeaders` will appear in the merged list.

In the input to the backend Lambda function, the `requestContext` object is a map of key-value pairs. In each pair, the key is the name of a $context variable property, and the value is the value of that property. API Gateway might add new keys to the map.

Depending on the features that are enabled, the `requestContext` map may vary from API to API. For example, in the preceding example, no authorization type is specified, so no `$context.authorizer.*` or `$context.identity.*` properties are present. When an authorization type is specified, this causes API Gateway to pass authorized user information to the integration endpoint in a `requestContext.identity` object as follows:

- When the authorization type is `AWS_IAM`, the authorized user information includes `$context.identity.*` properties.
- When the authorization type is `COGNITO_USER_POOLS` (Amazon Cognito authorizer), the authorized user information includes `$context.identity.cognito*` and `$context.authorizer.claims.*` properties.
- When the authorization type is `CUSTOM` (Lambda authorizer), the authorized user information includes `$context.authorizer.principalId` and other applicable `$context.authorizer.*` properties.

Output format of a Lambda function for proxy integration

In Lambda proxy integration, API Gateway requires the backend Lambda function to return output according to the following JSON format:

```
{
    "isBase64Encoded": true|false,
    "statusCode": httpStatusCode,
    "headers": { "headerName": "headerValue", ...
    "multiValueHeaders": { "headerName": ["headerV
    "body": "..."
}
```

In the output:

- The `headers` and `multiValueHeaders` keys can be unspecified if no extra response headers are to be returned.
- The `headers` key can only contain single-value headers.
- The `multiValueHeaders` key can contain multi-value headers as well as single-value headers. You can use the `multiValueHeaders` key to specify all of your extra headers, including any single-value ones.
- If you specify values for both `headers` and `multiValueHeaders`, API Gateway merges them into a single list. If the same key-value pair is specified in both, only the values from `multiValueHeaders` will appear in the merged list.

To enable CORS for the Lambda proxy integration, you must add `Access-Control-Allow-Origin: domain-name` to the output `headers`. *`domain-name`* can be `*` for any domain name. The output `body` is marshalled to the frontend as the method response payload. If `body` is a binary blob, you can encode it as a Base64-encoded string by setting `isBase64Encoded` to `true` and configuring `*/*` as a **Binary Media Type**. Otherwise, you can set it to `false` or leave it unspecified.

> **Note**
>
> For more information about enabling binary support, see Enabling binary support using the API Gateway console. For an example Lambda function, see Return binary media from a Lambda proxy integration.

If the function output is of a different format, API Gateway returns a `502 Bad Gateway` error response.

To return a response in a Lambda function in Node.js, you can use commands such as the following:

- To return a successful result, call `callback(null, {"statusCode": 200, "body": "results"})`.
- To throw an exception, call `callback(new Error('internal server error'))`.
- For a client-side error (if, for example, a required parameter is missing), you can call `callback(null, {"statusCode": 400, "body": "Missing parameters of ..."})` to return the error without throwing an exception.

In a Lambda `async` function in Node.js, the equivalent syntax would be:

- To return a successful result, call `return {"statusCode": 200, "body": "results"}`.
- To throw an exception, call `throw new Error("internal server error")`.
- For a client-side error (if, for example, a required parameter is missing), you can call `return {"statusCode": 400, "body": "Missing parameters of ..."}` to return the error without throwing an exception.

Next topic: Set up Lambda custom integrations

Previous topic: Lambda integration

- Connect with an AWS IQ expert