Quick Developer's Guide to Function-as-a-Service

Step 1: Choose technology

AWS Documentation » Amazon API Gateway » Developer Guide » Getting Started with Amazon API Gateway » Make Synchronous Calls to Lambda Functions

Make Synchronous Calls to Lambda Functions

AWS Lambda provides an easy way to build back-ends without managing servers. API Gateway and Lambda together provide a powerful platform for building web applications. This topic shows you how to create Lambda functions and build an API Gateway API to enable a Web client to call the Lambda functions synchronously. For more information about Lambda, see the AWS Lambda Developer Guide. For information about asynchronous invocation of Lambda functions, see Create an API as a Lambda Proxy.

Topics

- Step 1: Prerequisites
- Step 2: Create an API
- Step 3: Create a resource
- Step 4: Create Lambda Functions
- Step 5: Create and Test a GET Method
- Step 6: Create and Test a POST Method
- Step 7: Deploy the API
- Step 8: Test the API
- Step 9: Clean Up
- Next Steps
- Create Lambda Invocation and Execution Roles

Step 1: Prerequisites

You must grant API Gateway access permission to the IAM user who will perform the tasks discussed next. This IAM user must have full access to work with Lambda. For this, you can use or customize the managed policy of **AWSLambdaFullAccess**. `arn:aws:iam::aws:policy/AWSLambdaFullAccess` and attach it to a desired IAM user. For more information, see Get Ready to Use API Gateway. The IAM user must also be allowed to create policies and roles in order to complete the tasks discussed next. Attach the managed policy **IAMFullAccess** `arn:aws:iam::aws:policy/IAMFullAccess` and attach it to the desired IAM user.

Step 2: Create an API

In this step, you will create a new API named `myDemoAPI`. To create the new API, follow the steps in Build an API Gateway API Step by Step.

Step 3: Create a Resource

In this step, you will create a new resource named **/myDemoResource**. To create this resource, follow the steps in Build an API Gateway API Step by Step.

Step 4: Create Lambda Functions

> **Note**
> Creating Lambda functions may result in charges to your AWS account.

In this step, you will create two new Lambda functions. The first Lambda function, `GetHelloWorld`, is set up to call to Amazon CloudWatch and return the JSON object `"Hello": "World!"`. For more information about JSON, see Introducing JSON.

The second Lambda function, `Calculator`, will take an input ("name") to the call to CloudWatch, and return the JSON object `"Hello": <name>Logging<operation>`. If no input value is provided, the value will be "No Name".

You will use the Lambda console to create the Lambda functions and set up the required execution role/policy. You will then use the API Gateway console to create an API to integrate API methods with the Lambda functions, the API Gateway console to deploy the API, and the required API invocation role/policy. If you set up the API without using the API Gateway console, such as when Importing an API from Swagger, you must explicitly create, if necessary, and set up the required invocation role/policy. For more information about how to set up the invocation role/policy, see Create Lambda Invocation and Execution Roles, see Create Lambda Invocation and Execution Roles. For more information about Lambda see AWS Lambda Developer Guide.

To create the GetHelloWorld Lambda function

1. Open the AWS Lambda console at https://console.aws.amazon.com/lambda/.
2. Do one of the following:
   - If the welcome page appears, choose **Get Started Now**.
   - If the **Lambda: Function list** page appears, choose **Create a Lambda function.**
3. From **Select blueprint**, select the **hello-world** blueprint for `node.js`. You may need to type `hello` in the search filter to bring the blueprint in focus.
4. For **Name**, type **GetHelloWorld**.
5. For **Description**, type **Returns {"Hello": "World"}.**
6. For **Runtime**, choose **Node.js** or leave as-is.
7. Under **Lambda function code**, replace the default code statements in the inline code editor with the following:

```
'use strict';
console.log('Loading event');

exports.handler = function(event, context) {
    console.log("Hello:"+"World!"); // SUCCESS with message
    context.done(null, "Hello:"+"World!");
};
```

> **Tip**
> The preceding code is written in Node.js. The `console.log` method writes information to an Amazon CloudWatch Log. The `event` parameter contains the event's data. The `context` parameter contains callback context. Lambda calls `context.done` to perform follow-up actions. For more information about how to write Lambda function code, see the "Programming Model" section in AWS Lambda: How it Works and the sample walkthroughs in the AWS Lambda Guide.

8. Under **Lambda function handler and role**, leave the default of `index.handler` for **Handler.**
9. For **Role**, choose **Basic execution role**. **Create new role** appears.
   a. Leave the default selection of `Lambda_basic_execution` for **IAM Role.**
   b. Leave the default selection of `Create a new Role Policy` for **Policy Name.**
   c. Choose **Allow**.
10. For **Advanced settings** leave the default setting as is.
11. Choose **Next**.
12. Choose **Create function.**
13. For the newly created **GetHelloWorld** function, note the AWS region where you created this function. You will need it later.
14. To test the newly created function, as a good practice, choose **Actions** and then select **Configure test event**.
15. For **Input test event**, replace any default code statements with the following, and then choose **Save and test**:

```
{}
```

> **Tip**
> This function does not use any input. Therefore, we provide an empty JSON object as the input.

16. Choose **Test** to invoke the function. The **Execution result** section shows `"Hello": "World!"`. The output is also written to CloudWatch Logs.
17. Go to the **Functions** list to create the next Lambda function.

In addition to the Lambda function, an IAM role (`Lambda_basic_execution`) is also created as the result of this procedure. You can view this in the IAM console. Attached to this IAM role is the following inline policy that grants users of your AWS account permission to call the CloudWatch `CreateLogGroup`, `CreateLogStream`, and `PutLogEvents` actions on any of the CloudWatch resources.

```
{
  "Statement": [{
    "Effect": "Allow",
    "Action": [
      "logs:CreateLogGroup",
      "logs:CreateLogStream",
      "logs:PutLogEvents"
    ],
    "Resource": "arn:aws:logs:*:*:*"
  }]
}
```

A trusted entity of this IAM role is `lambda.amazonaws.com`, which has the following trust relationship:

```
{
  "Statement": [{
    "Sid": "",
    "Effect": "Allow",
    "Principal": {
      "Service": "lambda.amazonaws.com"
    },
    "Action": "sts:AssumeRole"
  }]
}
```

The combination of this trust relationship and the inline policy makes it possible for the user to invoke the Lambda function and for Lambda to call the supported CloudWatch actions on the user's behalf.

To create the GetHelloWithName Lambda function

1. Choose **Create a Lambda function.**
2. From **Select blueprint**, select the **hello-world** blueprint for `node.js`.
3. Type **GetHelloWithName** for **Name**.
4. For **Description**, type **Returns {"Hello:", a user-provided string, and "}.**
5. For **Runtime**, choose **Node.js**.
6. In the code editor under **Lambda function code** replace the default code statements with the following:

```
'use strict';
console.log('Loading event');

exports.handler = function(event, context) {
    var name = event.name === undefined ? 'No-Name' : event.name;
    console.log("Hello:"+" + name + "!");
    context.done(null, "Hello:"+name+"!");  // SUCCESS with message
};
```

7. Under **Lambda function handler and role**, leave the default of `index.handler` for **Handler.**
8. For **Role**, choose **Lambda_basic_execution** under **Use existing role**, assuming you have created the `Lambda_basic_execution` role in the previous procedure.
9. Leave the default values for **Advanced settings**. Then choose **Next**.
10. Choose **Create Function.**
11. For the newly created **GetHelloWithName** function, note the AWS region where you created this function. You will need it in later steps.
12. To test this newly created function, choose **Actions** and then **Configure test event**.
13. In **Input test event**, replace any default code statements with the following, and then choose **Save and test**:

```
{
  "name": "User"
}
```

> **Tip**
> The function calls `context.name` to read the input name. We expect it to return `"Hello", "User!"`, given the above input.
>
> You can experiment with this function by removing `"name": "User"` from the **Input test event** for the function and choosing **Save and test** again. You should see the output of `"Hello", "No-Name"` under **Execution result** in the Lambda console, as well as in CloudWatch Logs.

Step 5: Create and Test a GET Method

Switch back to the API Gateway console. In this step, you will create a GET method, connect it to your `GetHelloWorld` function in Lambda, and then test it. You use a GET method primarily to retrieve or read a representation of a resource. If successful, the GET method will return a JSON-formatted object.

To create and test the GET method

1. In the API Gateway console, from **APIs**, choose **MyDemoAPI**.
2. In the **Resources** pane, choose **/mydemoresource**. From **Actions**, choose **Create Method**. Choose GET from the HTTP method drop-down list and then choose the checkmark to create the method.
3. In the GET method **Setup** pane, for **Integration type**, choose **Lambda Function**. For **Lambda Region**, choose the region (e.g. `us-east-1`) where you created the Lambda functions. In **Lambda Function**, type `GetHelloWorld`. Choose **Save** to finish setting up the integration request for this method.

For a list of region names and identifiers, see AWS Lambda in the *Amazon Web Services General Reference*.

4. For **Add Permission to Lambda Function**, choose **OK**.
5. In the **Method Execution** pane, choose **TEST** from the **Client** box, and then choose the **Test** button. If successful, **Response Body** will display the following:

```
{
  "Hello": "World!"
}
```

By default, API Gateway will pass through the request from the API caller. For the GET method call you just created, API Gateway will not pass any data to the Lambda function because an empty JSON response by default and then return the response from the Lambda function without modifications.

In the next steps, you will create a POST method call. For POST and PUT method calls, you can pass in a request body in JSON format, which the Lambda function will receive as its input event. Optionally, you can transform the input to the Lambda function by using mapping templates in API Gateway.

Step 6: Create and Test a POST Method

In this step, you will create a new POST method, connect it to your `GetHelloWithName` function in Lambda, and then test it. You use a POST method primarily to send data to the server for creating a new resource. In this walkthrough, the POST method will simply return a JSON-formatted object.

To create and test the POST method

1. In the **Resources** pane, choose **/mydemoresource**, and then choose **Create Method**.
2. For the HTTP method, choose **POST**, and then choose the checkmark to save your choice.
3. In the **Setup** pane, for **Integration Type**, choose **Lambda Function**.
4. For **Lambda Region**, choose the region identifier that corresponds to the region name in which you created the `GetHelloWithName` Lambda function.
5. For **Lambda Function**, type **GetHelloWithName**, and then choose **Save**.
6. When you are prompted to give API Gateway permission to invoke your Lambda function, choose **OK**.
7. In the **Method Execution** pane, in the **Client** box, and then choose **TEST**. Expand **Request Body**, and type the following:

```
{
  "name": "User"
}
```

8. Choose **Test**. If successful, **Response Body** will display the following:

```
{
  "Hello": "User!"
}
```

9. Change **Request Body** by removing **"name": "User"** so that only a set of curly brackets `{}` remain, and then choose **Test** again. If successful, **Response Body** will display the following:

```
{
  "Hello": "No-Name"
}
```

You can also test the POST method using a command-line tool, such as cURL or Postman. For the cURL command-line tool, you can call `curl -v -X POST https://{restapi-id}.execute-api.{region}.amazonaws.com/{stage-name}/{resource-name} -H 'content-type:application/json' -d '{"name": "User"}'` to test the POST method. For the Postman tool, you can set the request body to `{"name": "User"}` and the header to `content-type:application/json`.

The GET and POST methods discussed here are both integrated with a `node` request in the back end:

```
POST /2015-03-31/functions/myfunction/invocations?Qualifier=prod HTTP/1.1
X-Amz-Invocation-Type: RequestResponse

Content-Type: application/json
Content-Length: 16\n\n{"key1":"1"}
```

The X-Amz-Invocation-Type: RequestResponse header specifies that the Lambda function be invoked synchronously. The request body, `{"key1":"1"}`, is an example input to the Lambda function in JSON format. In this walkthrough, the console sets `myfunction` as **GetHelloWorld** for the GET method request and supplies an empty JSON payload when you invoke the method. For the POST method, the console sets `myfunction` as **GetHelloWithName** and passes the caller-supplied method request's JSON payload as-is. You can configure API Gateway to regain full control of a method creation and setup by going through the AWS service proxy integration directly. For more information, see Create an API as a Lambda Proxy.

Step 7: Deploy the API

You are now ready to deploy your API so that you can call it outside of the API Gateway console. In this step, you will create a stage, `k-API-Gateway`, a stage defines the path through which an API deployment is accessible. For example, you can define a `test` stage and deploy your API to it, so that a resource named `myDemoGET` is accessible through a URL that ends in `/test/myDemoGET`.

To deploy the API

1. Choose the API from the **APIs** pane or choose a resource or method from the **Resources** pane. Choose **Deploy API** from the **Actions** drop-down menu.
2. For **Deployment stage**, choose **New Stage.**
3. For **Stage name**, type **test.**

   > **Note**
   > The input must be UTF-8 encoded (i.e., unlocalized) text.

4. For **Stage description**, type **this is a test.**
5. For **Deployment description**, type **Calling Lambda functions walkthrough.**
6. Choose **Deploy**.

Step 8: Test the API

In this step, you will go outside of the API Gateway console to call the GET and POST methods in the API you just deployed.

To test the GET on mydemoresource method

1. In the **Stage Editor** pane, copy the URL from **Invoke URL** to the clipboard. It should look something like this:

   `https://{restapi-id}.execute-api.{region}.amazonaws.com/test`

2. In a separate web browser tab or window, paste the URL into the address box. Append the path to your resource's (**/mydemoresource**) to the end of the URL. The URL should look something like this:

   `https://{restapi-id}.execute-api.{region}.amazonaws.com/test/mydemoresource`

3. Browse to this URL. If the GET method is successfully called, the web page will display:

   `{"Hello":"World!"}`

To test the POST on mydemoresource method

1. You will not be able to test a POST method request with your web browser's address bar. Instead, use an advanced REST API client, such as Postman, or the cURL command-line tool.
2. Send a POST method request to the URL from the previous procedure. The URL should look something like this:

   `https://{restapi-id}.execute-api.{region}.amazonaws.com/test/mydemoresource`

   Be sure to append to the request headers the following header:

   `Content-Type: application/json`

   Also, be sure to add the following code to the request body:

   ```
   {
     "name": "User"
   }
   ```

   For example, if you use the cURL command-line tool, run a command similar to the following:

   ```
   curl -X POST https://{restapi-id}.execute-api.{region}.amazonaws.com/test/mydemoresource -H 'content-type:application/json' -d '{"name": "User"}'
   ```

   If the POST method is successfully called, the response should contain:

   `{"Hello":"User!"}`

Step 9: Clean Up

If you no longer need the Lambda functions you created for this walkthrough, you can delete them now. You can also delete the accompanying IAM resources.

> **Caution**
> If you plan to complete the other walkthroughs in this series, do not delete the Lambda execution role or the Lambda invocation role. If you delete a Lambda function that your APIs rely on, those APIs will no longer work. Deleting a Lambda function could also affect your ability to use the Lambda function console effectively in the future.
>
> If you delete an IAM resource that a Lambda function relies on, that Lambda function will no longer work, and any APIs that rely on that function will no longer work. Deleting an IAM resource is irreversible. If you want to use the IAM resource again, you must re-create the resource.

To delete the Lambda functions

1. Sign in to the AWS Management Console and open the AWS Lambda console at https://console.aws.amazon.com/lambda/.
2. From the list of functions, choose **GetHelloWorld**, choose **Actions** and then choose **Delete function.** When prompted, choose **Delete**.
3. From the list of functions, choose **GetHelloWithName**, choose **Actions**, and then choose **Delete function.** When prompted, choose **Delete** again.

To delete the associated IAM resources

1. Open the Identity and Access Management (IAM) console at https://console.aws.amazon.com/iam/.
2. From **Details**, choose **Roles.**
3. From the list of roles, choose **APIGatewayLambdaExecRole**, choose **Role Actions** and then choose **Delete Role.** When prompted, choose **Yes, Delete.**
4. From **Details**, choose **Policies.**
5. From the list of policies, choose **APIGatewayLambdaExecPolicy**, choose **Policy Actions** and then choose **Delete.** When prompted, choose **Delete**.

You have now reached the end of this walkthrough.

Next Steps

You may want to proceed to the next walkthrough, which shows how to map header parameters from the method request to the integration request and from the integration response to the method response. It uses the HTTP proxy integration to connect your API to HTTP endpoints in the back end.

For more information about API Gateway, see What Is Amazon API Gateway?. For more information about REST, see REST(ful) Web Services: A Tutorial.

Auth0 Webtasks
https://webtask.io

1. Install wt command line interface.
wt is an open source Node.js CLI to interact with the webtask API.

```
npm install -g wt-cli
```

2. Initialize wt.
wt will ask for an e-mail or phone number to send you an activation code.

```
wt init tomek@pinczuk.org
```

3. Create a simple webtask.
Now let's make our first webtask, simply returning text/plain "hello, webtask".

```
echo 'module.exports = function (cb) {cb(null, "Hello.")}' > hello.js
wt create hello.js
```

Congratulations! You have just created your first webtask.
https://webtask.it.auth0.com/api/run/wt-tomasz-pinczuk_org-0/hello?webtask_no_cache=1

1. There is no four
5. Go home
6. Spend time with family
7. Read a book
8. Play soccer
9. Walk the dog
10. Plant a tree
11. Learn Erlang
12. Pick up a new hobby
13. Start a business
14. Just lazy around
15. Hike a trail
16. Write a blog
17. Cook dinner
18. Plant another tree
19. Invite friends over
20. Go sailing
21. Play chess
22. Prune the garden
23. Cook asado
24. Do some knitting
25. Visit friends
26. Learn to tap dance
27. Interview for a job with Auth0

Just have a life already

https://webtask.io

Auth0