FaaS – Development of a serverless web service with AWS Lambda

**Objective:** Get practical experience in development and deployment of a serverless functions on Cloud FaaS with AWS Lambda.

**Tasks:**

1. Create a Calculator Lambda function
2. Configure permissions and trust relations
3. Test the Calculator Lambda function
4. Create the Calculator Web API
5. Create HTTP methods to call the Lambda Function
6. Deploy and testing the API
7. Un-deploy the API

**Lab environment**:

* SoapUI tool

*Contents*

[Step 0: Preparation and Logging to AWS management console 3](#_heading=h.30j0zll)

[Step 1: Creating a Calculator Lambda function 4](#_heading=h.1fob9te)

[Step 2: Configuring permissions and trust relations 7](#_heading=h.3znysh7)

[Step 3: Testing the Calculator Lambda function 12](#_heading=h.2et92p0)

[Step 4: Creating a Web API for the Calculator Lambda function 14](#_heading=h.tyjcwt)

[Step 5: Integration – Creating HTTP methods to call the Lambda Function 15](#_heading=h.3dy6vkm)

[Create a GET method with query parameters to call the Lambda function 15](#_heading=h.1t3h5sf)

[Create a POST method with a JSON payload to call the Lambda function 21](#_heading=h.4d34og8)

[Create a GET method with path parameters to call the Lambda function 21](#_heading=h.2s8eyo1)

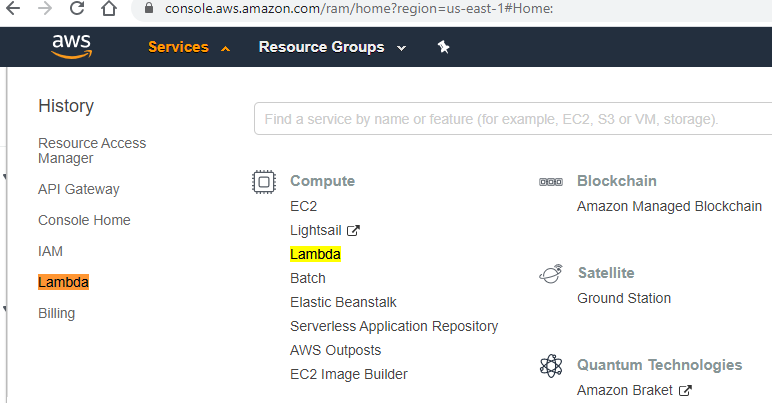
[Step 6: Deploying and testing the API 22](#_heading=h.17dp8vu)

[Step 7: Un-deploying the API 23](#_heading=h.26in1rg)

# Step 0: Preparation and Logging to AWS management console

You'll create a Lambda function using the AWS console.

1. Sign into your AWS account and open AWS Console: <https://console.aws.amazon.com/>
2. Open AWS Lambda Console



# 

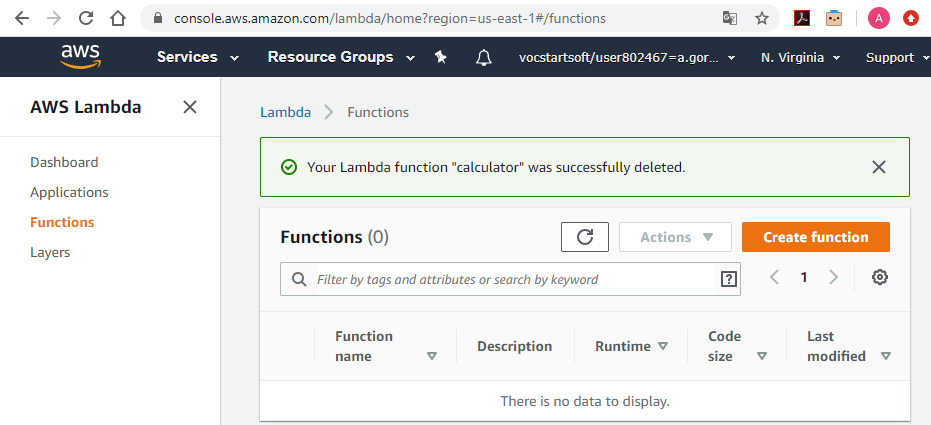
# Step 1: Creating a Calculator Lambda function

AWS Lambda is an event-driven, serverless computing platform provided by Amazon as a part of Amazon Web Services. It is a computing service that runs code in response to events and automatically manages the computing resources required by that code.

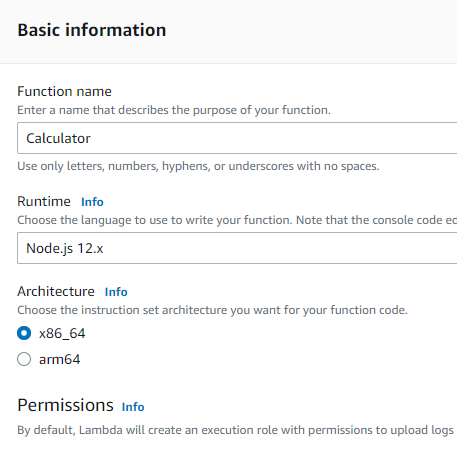
The purpose of Lambda, as compared to AWS EC2, is to simplify building smaller, on-demand applications that are responsive to events and new information. AWS targets starting a Lambda instance within milliseconds of an event. AWS Lambda officially supports Node.js, Python, Java, Go,[2] Ruby,[3] and C# (through .NET Core).

Unlike Amazon EC2, which is priced by the hour but metered by the second, AWS Lambda is metered in increments of 100 milliseconds. Usage amounts below a documented threshold fall within the AWS Lambda free tier - which does not expire 12 months after account signup, unlike the free tier for other AWS services.

1. In the AWS Lambda console, choose **Create function**

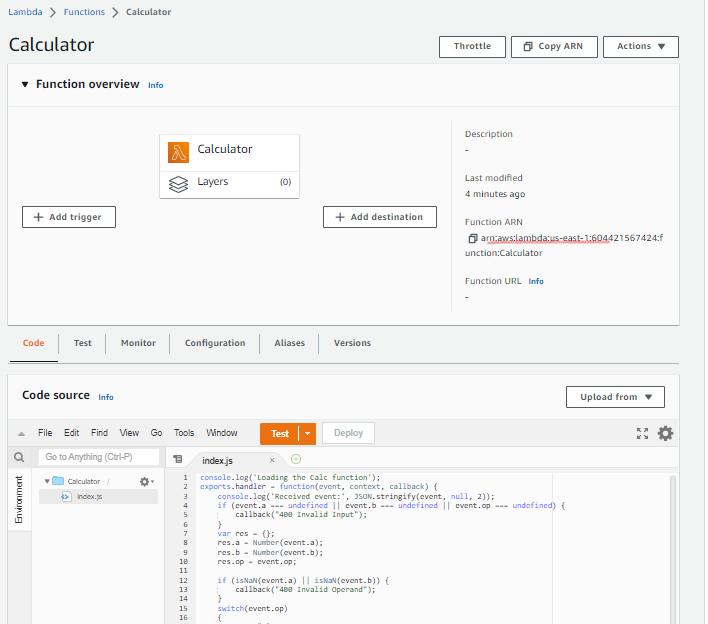


1. Choose **Author from Scratch**.
2. For **Name**, enter **Calculator**.
3. Set the **Runtime** to Node.js 12.x (use 12.x version).
4. Set **Architecture** to x86\_64.
5. Choose **Create function**.



1. Paste (use MS Word to copy/paste) the following Lambda function and paste it into the code editor in the Lambda console.

| console.log('Loading the Calc function');  exports.handler = function(event, context, callback) {  console.log('Received event:', JSON.stringify(event, null, 2));  if (event.a === undefined || event.b === undefined || event.op === undefined) {  callback("400 Invalid Input");  }  var res = {};  res.a = Number(event.a);  res.b = Number(event.b);  res.op = event.op;    if (isNaN(event.a) || isNaN(event.b)) {  callback("400 Invalid Operand");  }  switch(event.op)  {  case "+":  case "add":  res.c = res.a + res.b;  break;  case "-":  case "sub":  res.c = res.a - res.b;  break;  case "\*":  case "mul":  res.c = res.a \* res.b;  break;  case "/":  case "div":  res.c = res.b===0 ? NaN : Number(event.a) / Number(event.b);  break;  default:  callback("400 Invalid Operator");  break;  }  callback(null, res);  } |
| --- |



1. Notice the ARN (Amazon Resource Locator) of the created function on the top of the page
2. Choose **Deploy**.

# Step 3: Testing the Calculator Lambda function

Created function requires two operands (a and b) and an operator (op) from the event input parameter. The input is a JSON object of the following format:

{

"a": "Number" | "String",

"b": "Number" | "String",

"op": "String"

}

This function returns the calculated result (c) and the input. For an invalid input, the function returns either the null value or the "Invalid op" string as the result. The output is of the following JSON format:

{

"a": "Number",

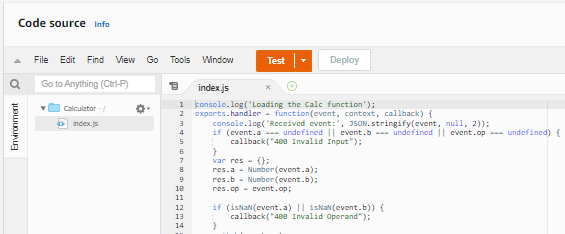
"b": "Number",

"op": "String",

"c": "Number" | "String"

}

You should test the function in the Lambda console before integrating it with the API in the next step. Here's how to test your Calculator function in the Lambda console:



1. Click on **Test** and create a new test event; for the test event name enter **calc2plus5**.
2. Replace the test event definition with the following:

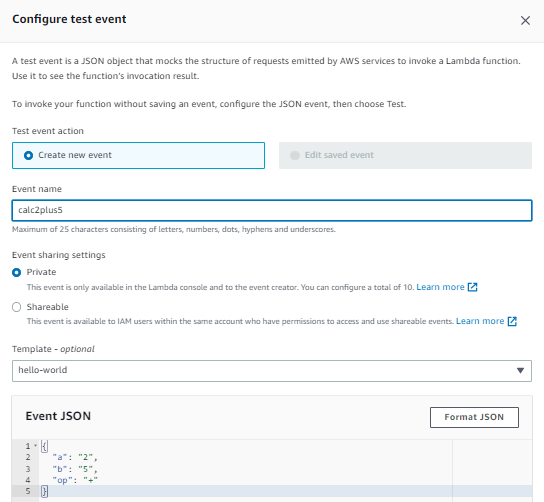
{

"a": "2",

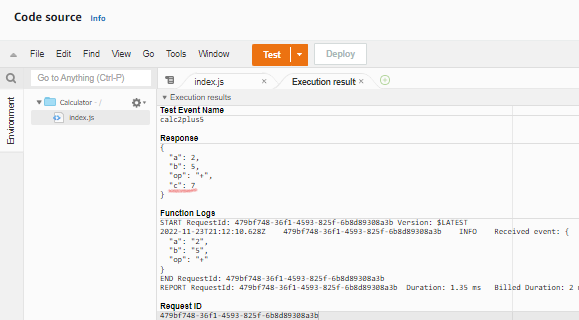
"b": "5",

"op": "+"

}



1. Choose **Save**.
2. In the dropdown menu, choose **calc2plus5** and press **Test**.
3. Expand **Execution result: succeeded**. You should see the following:



# Step 3: Configuring permissions and trust relations

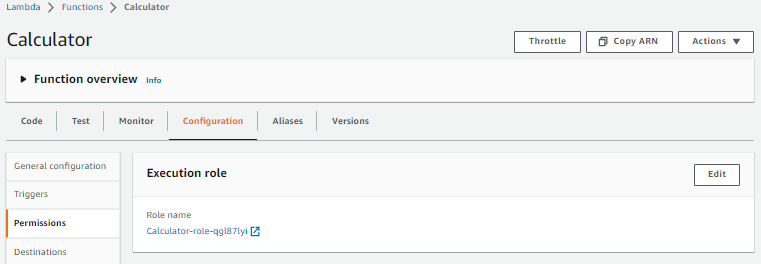
AWS **Identity and Access Management** (**IAM**) enables users to manage access to their AWS services and resources securely. Using IAM, you can create and manage AWS users and groups, and use permissions to allow and deny their access to AWS resources.

An **Execution Role** is an IAM role that Lambda has permission to assume when you invoke a function. An AWS Lambda function's **Execution Role** grants it permission to access AWS services and resources. The role is created once you create a function. You can also create an execution role in the IAM console. You can restrict the scope of a user's permissions by specifying resources and conditions in an **IAM policy** which is applied to function's **Execution Role**.

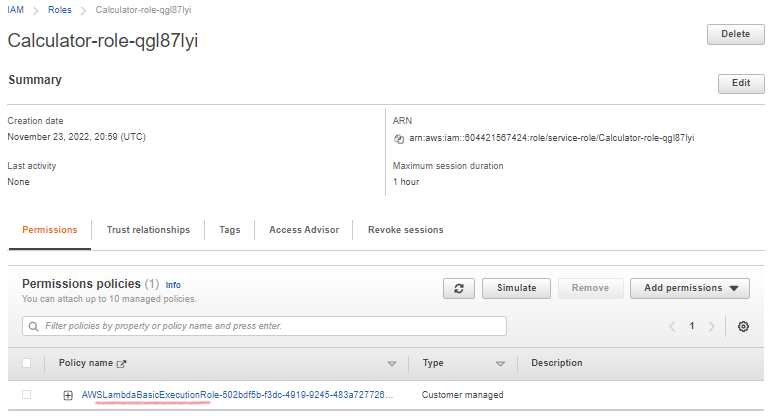
You can restrict the resources that a user can access by specifying the **Amazon Resource Name** (**ARN**) of a resource, or an ARN pattern that matches multiple resources. For example, you can create and apply a policy which allows the specified user (e.g. with the user account 123456789012) to invoke the specified function (e.g. named my-function) in the specified location (e.g. in US West Oregon Region) by specifying the following ARN: "arn:aws:lambda:us-west-2:123456789012:function:my-function".

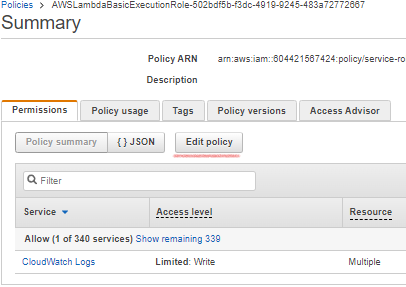
When the action doesn't act on a named resource, or when you grant permission to perform the action on all resources, the value of the resource in the policy is a wildcard (\*). Read more about AWS Access Management at: <https://docs.aws.amazon.com/IAM/latest/UserGuide/access.html>

1. Proceed to the **Configuration** -> **Permission** tab and choose the role created by default



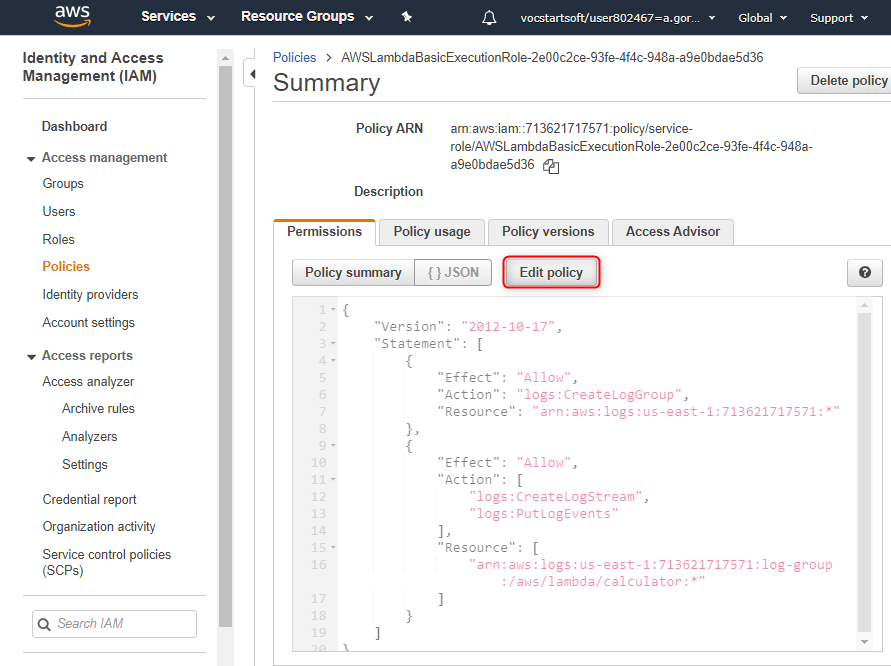
2. You will be redirected to the IAM console where you can specify role’s permissions and trust relations.





3. Replace the default permission policy with the new one.

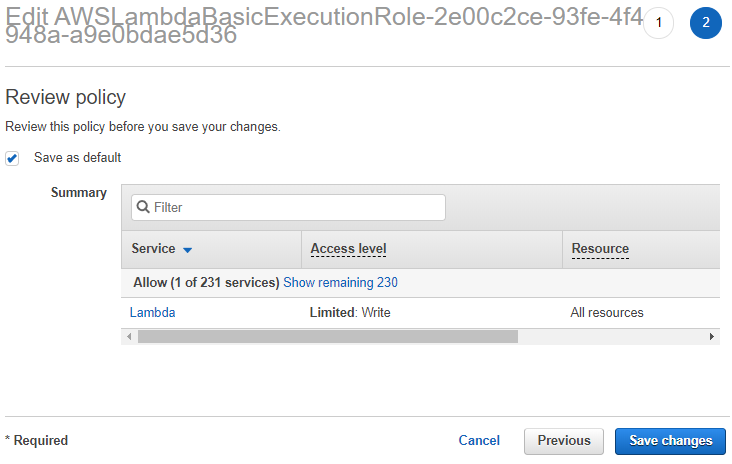
By default, Lambda creates an execution role with minimal permissions when you create a function in the Lambda console. In this scenario we would like to edit role’s policy to simply allow all users to invoke all functions in all locations.



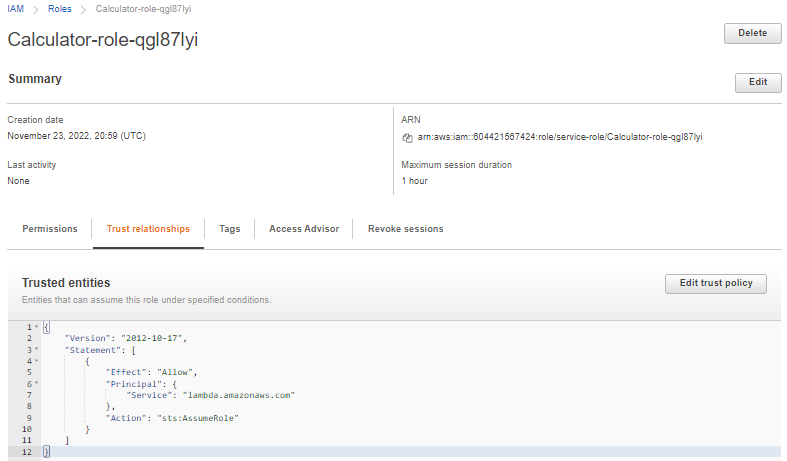
* 1. In the **JSON** tab, replace the existing policy with the following:

| {  "Version": "2012-10-17",  "Statement": [  {  "Effect": "Allow",  "Action": "lambda:InvokeFunction",  "Resource": "\*"  }  ]  } |  |
| --- | --- |

* 1. Choose **Review policy** and **Save** changes and close this tab.

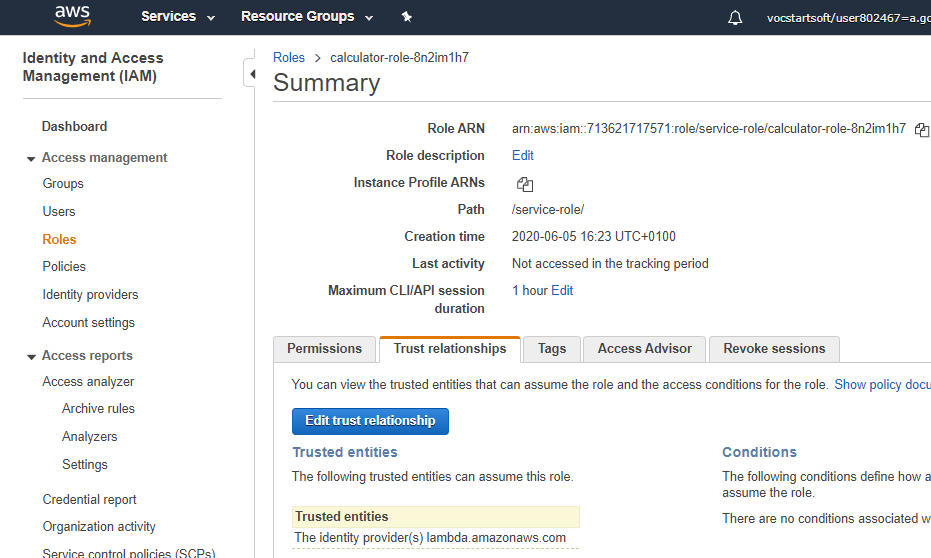


4. In the Roles settings select role’s **Trust relationships** tab



**Trust policy** is a JSON policy document in which you define the principals that you trust to assume the role. A role trust policy is a required resource-based policy that is attached to a role in IAM, and is one-half of the permissions. The other half is a permissions policy attached to the user in the trusted account that allows that user to switch to, or assume the role. The principals that you can specify in the trust policy include users, roles, accounts, and services.

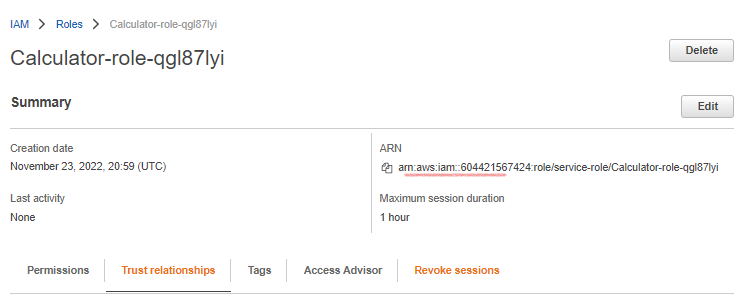
1. Choose the **Trust relationships** tab.
2. Choose **Edit trust relationship**.



1. Replace the existing policy with the following:

| {  "Version": "2012-10-17",  "Statement": [  {  "Sid": "",  "Effect": "Allow",  "Principal": {  "Service": [  "apigateway.amazonaws.com",  "lambda.amazonaws.com"  ]  },  "Action": "sts:AssumeRole"  }  ]  } |  |
| --- | --- |

5. Make a note of the role ARN for the role you just created. You'll need it later.



6. Make sure you have saved ARNs for the created Lambda function and its IAM role, e.g.:

* ***Function ARN***: arn:aws:lambda:us-east-1:604421567424:function:Calculator
* ***Calculator-role-qgl87lyi***: arn:aws:iam::604421567424:role/service-role/Calculator-role-qgl87lyi

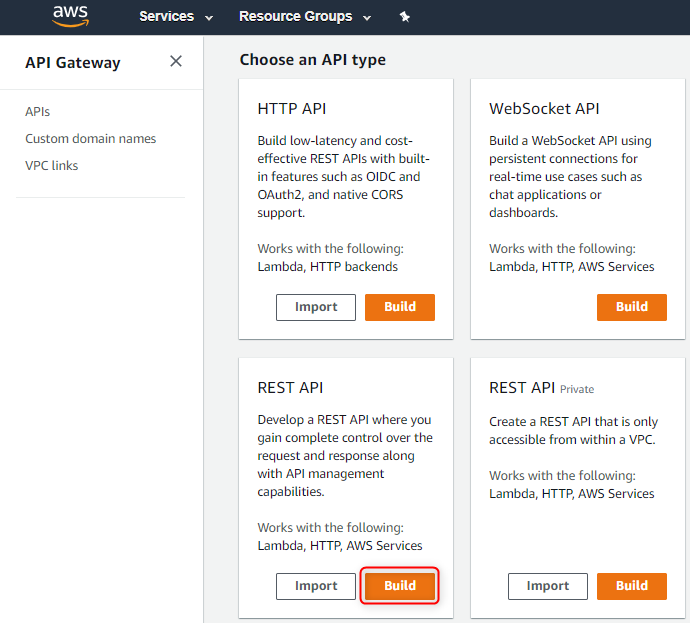
# Step 4: Creating a Web API for the Calculator Lambda function

Developers can create a web API with an HTTP endpoint for their Lambda functions by using Amazon API Gateway. Web APIs route HTTP requests to Lambda functions. API Gateway provides tools for creating and documenting web APIs. Developers can also secure access to your API with authentication and authorization controls.

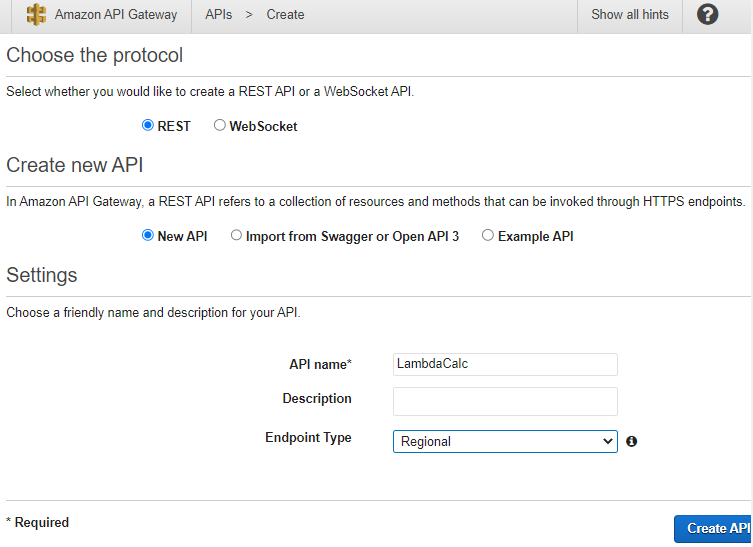
The following procedure shows how to create an API for the Calculator Lambda function you just created. In subsequent sections, you'll add resources and methods to it.

1. Sign in to the API Gateway console at <https://console.aws.amazon.com/apigateway>.
2. If this is your first time using API Gateway, you see a page that introduces you to the features of the service. Under **REST API**, choose **Build**. When the **Create Example API** popup appears, choose **OK**.

If this is not your first time using API Gateway, choose **Create API**. Under **REST API**, choose **Build**.



1. Under **Create new API**, choose **New API**.
2. For **API Name**, enter **LambdaCalc**.
3. Leave the **Description** blank, and leave the **Endpoint Type** set to **Regional**.
4. Choose **Create API**.

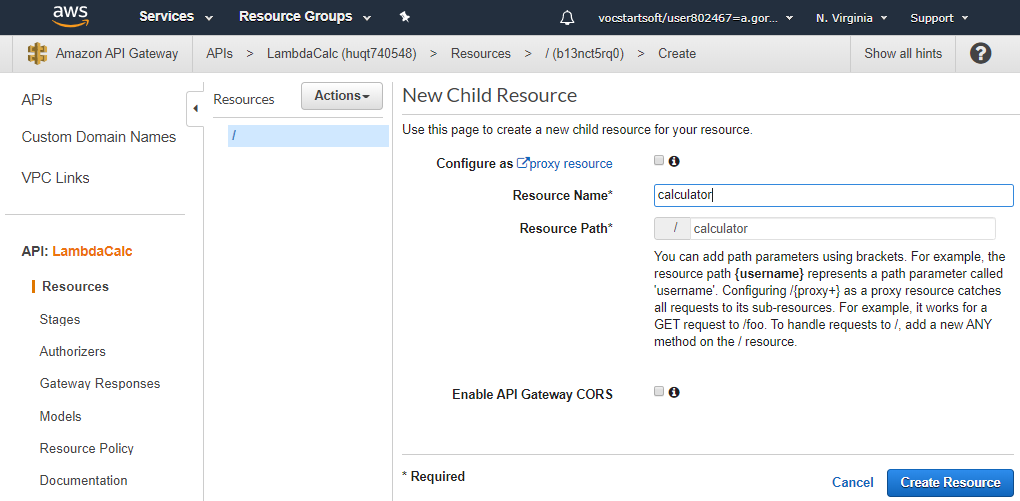


# Step 5: Integration – Creating HTTP methods to call the Lambda Function

## Create a GET method with query parameters to call the Lambda function

By creating a GET method that passes query string parameters to the Lambda function, you enable the API to be invoked from a browser. This approach can be useful, especially for APIs that allow open access. To set up the GET method with query string parameters do the following.

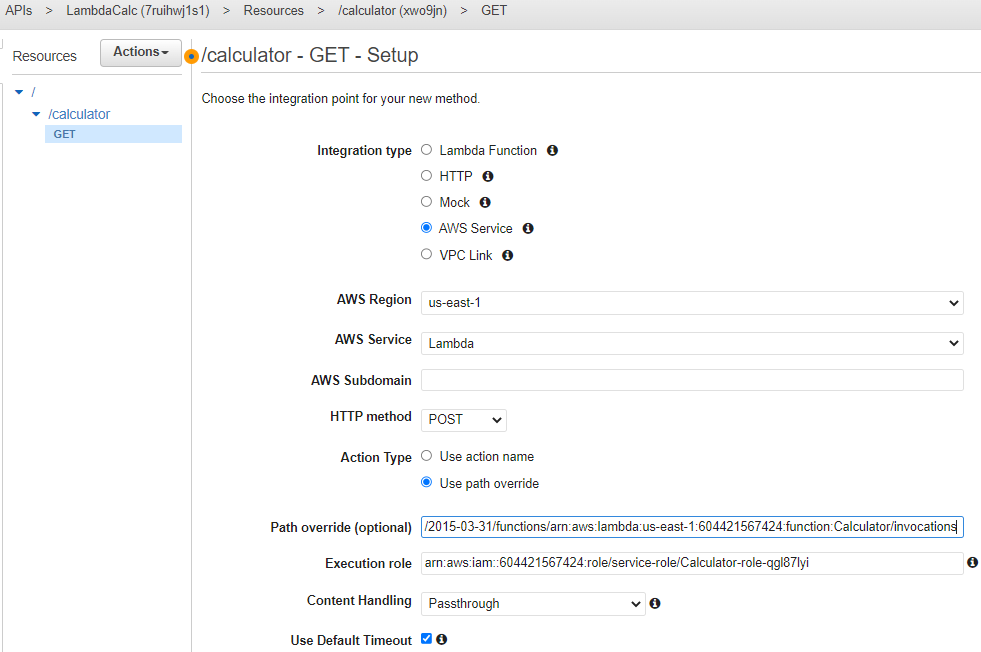
1. In the API Gateway console, under your LambdaCalc API's **Resources**, choose **/**.
2. In the **Actions** drop-down menu, choose **Create Resource**.
3. For **Resource Name**, enter **calculator**.
4. Choose **Create Resource**.



1. Choose the **/calculator** resource you just created.
2. In the **Actions** drop-down menu, choose **Create Method**.
3. In the method drop-down menu that appears, choose **GET**.
4. Choose the checkmark icon to save your choice.
5. In the **Set up** pane:
   1. For **Integration type**, choose **AWS Service**.
   2. For **AWS Region**, choose the region (e.g., us-east-1) where you created the Lambda function - check function’s ARN if not sure.
   3. For **AWS Service**, choose **Lambda**.
   4. Leave **AWS Subdomain** blank, because our Lambda function is not hosted on any AWS subdomain.
   5. For **HTTP method**, choose **POST** and choose the checkmark icon to save your choice.

*Lambda requires that the POST request be used to invoke any Lambda function. This example shows that the HTTP method in a frontend method request can be different from the integration request in the backend.*

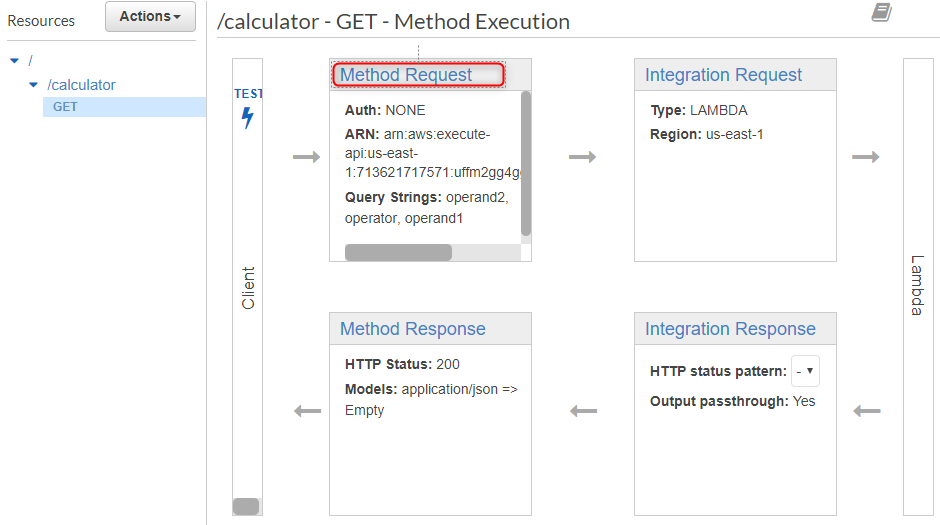
* 1. Choose Use path override for **Action Type**. This option allows us to specify the ARN of the invoke action to execute our Calculator function.
  2. Enter **/2015-03-31/functions/arn:aws:lambda:*region*:*account-id*:function:Calculator/invocations** in **Path override**, where *region* is the region where you created your Lambda function and *account-id* is the account number for the AWS account (you can notice region:account-id from the Function’s ARN at a time it was created – see Step 1).
  3. For **Execution role**, enter the role ARN for the **IAM role** you created earlier.
  4. Leave **Content Handling** set to **Passthrough**, because this method will not deal with any binary data.
  5. Leave **Use default timeout** checked.



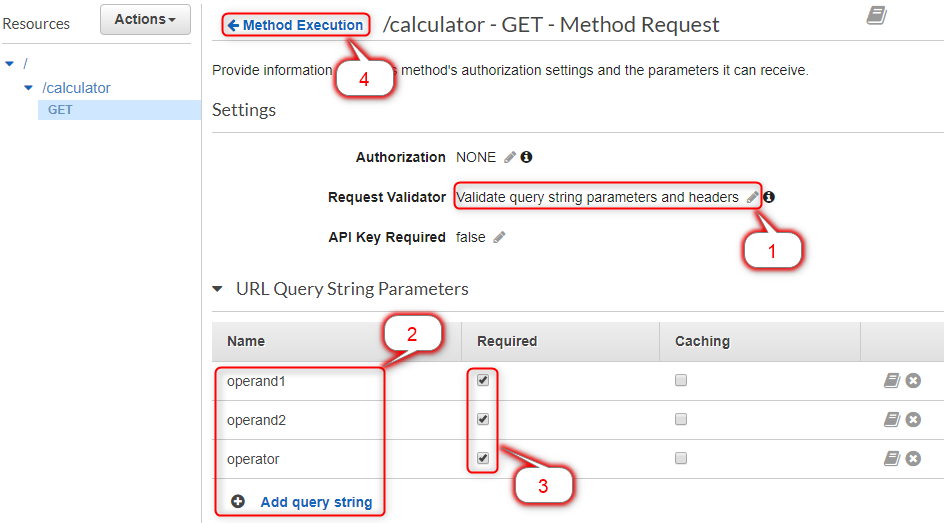
* 1. Choose **Save**.

1. Choose **Method Request**.

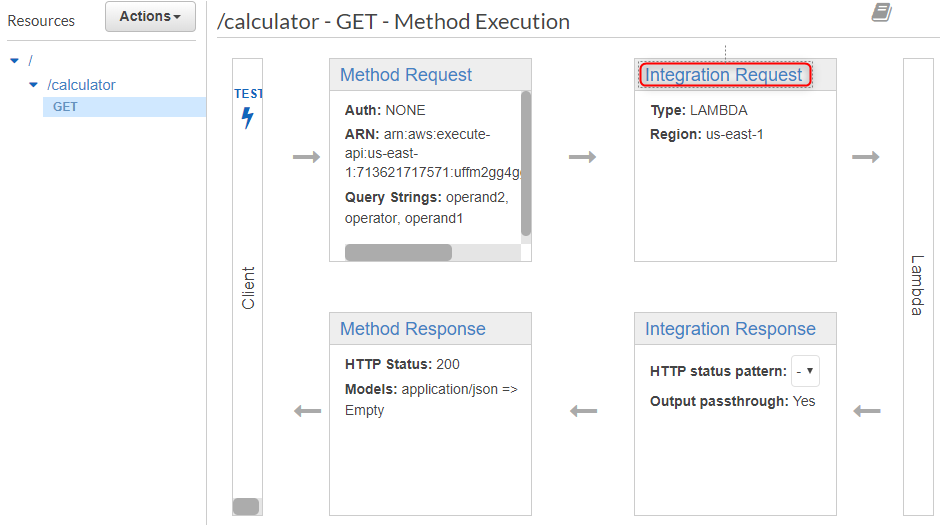
Now you will set up query parameters for the **GET** method on **/calculator** so it can receive input on behalf of the backend Lambda function.



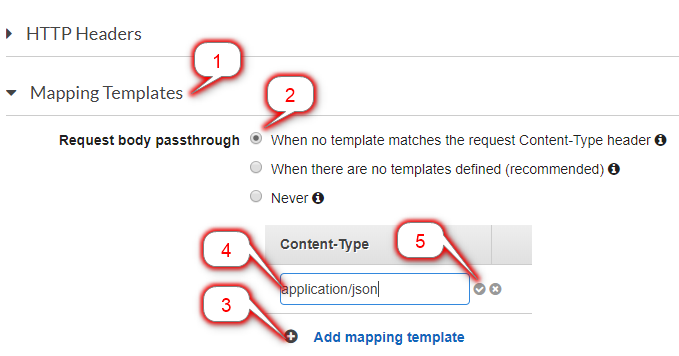
* 1. Choose the pencil icon next to **Request Validator** and choose **Validate query string parameters and headers** from the drop-down menu. This setting will cause an error message to return to state the required parameters are missing if the client does not specify them. You will not get charged for the call to the backend.
  2. Choose the checkmark icon to save your changes.
  3. Expand the **URL Query String Parameters** section.
  4. Choose **Add query string**.
  5. For **Name**, type **operand1**.
  6. Choose the checkmark icon to save the parameter.
  7. Repeat the previous steps to create parameters named **operand2** and **operator**.
  8. Check the **Required** option for each parameter to ensure that they are validated.



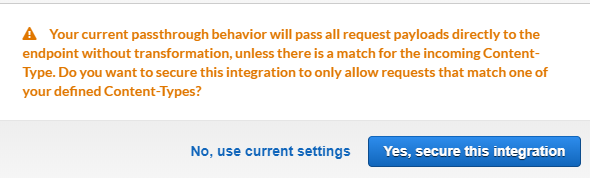
1. Choose **Method Execution** at the top of the page and then choose **Integration Request** to set up the mapping template to translate the client-supplied query strings to the integration request payload as required by the Calc function.



* 1. Expand the **Mapping Templates** section.
  2. Choose **When no template matches the request Content-Type header** for **Request body passthrough**.
  3. Under **Content-Type**, choose **Add mapping template**.
  4. Type **application/json** and choose the checkmark icon to open the template editor.



* 1. Choose **Yes, secure this integration** to proceed.



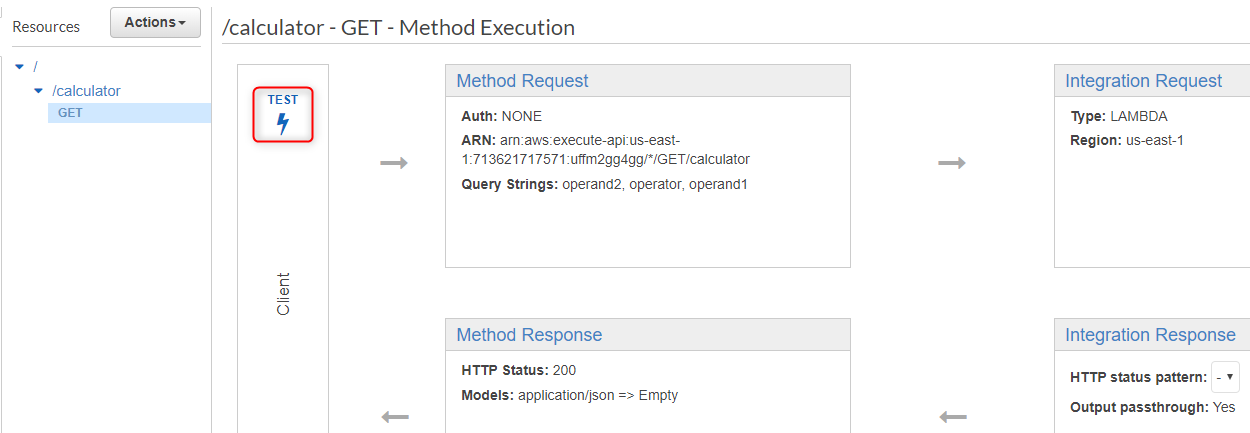
* 1. Copy the following mapping script into the mapping template editor:

| {  "a": "$input.params('operand1')",  "b": "$input.params('operand2')",  "op": "$input.params('operator')"  } |  |
| --- | --- |

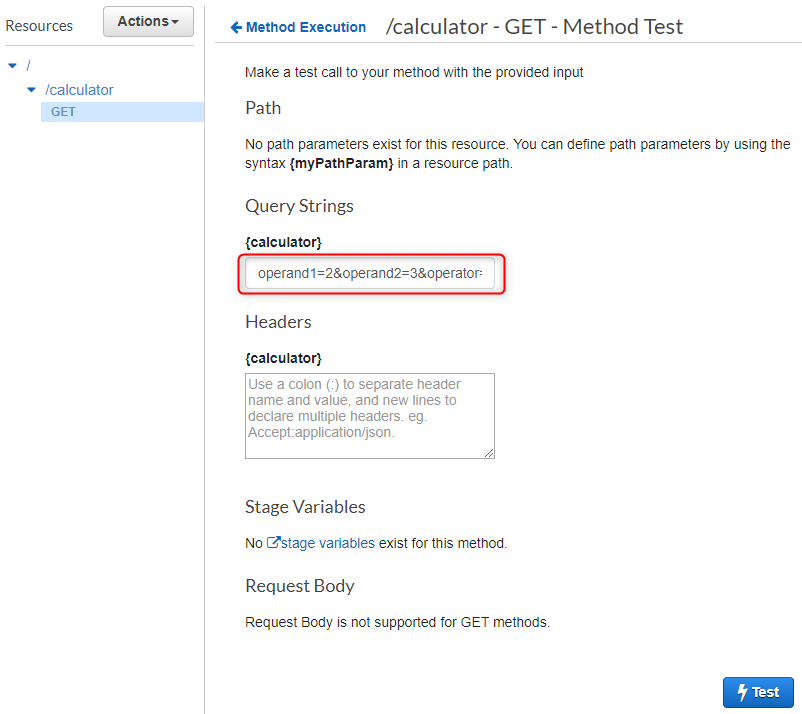
*This template maps the three query string parameters declared in****Method Request****into designated property values of the JSON object as the input to the backend Lambda function. The transformed JSON object will be included as the integration request payload.*

* 1. Choose **Save**.

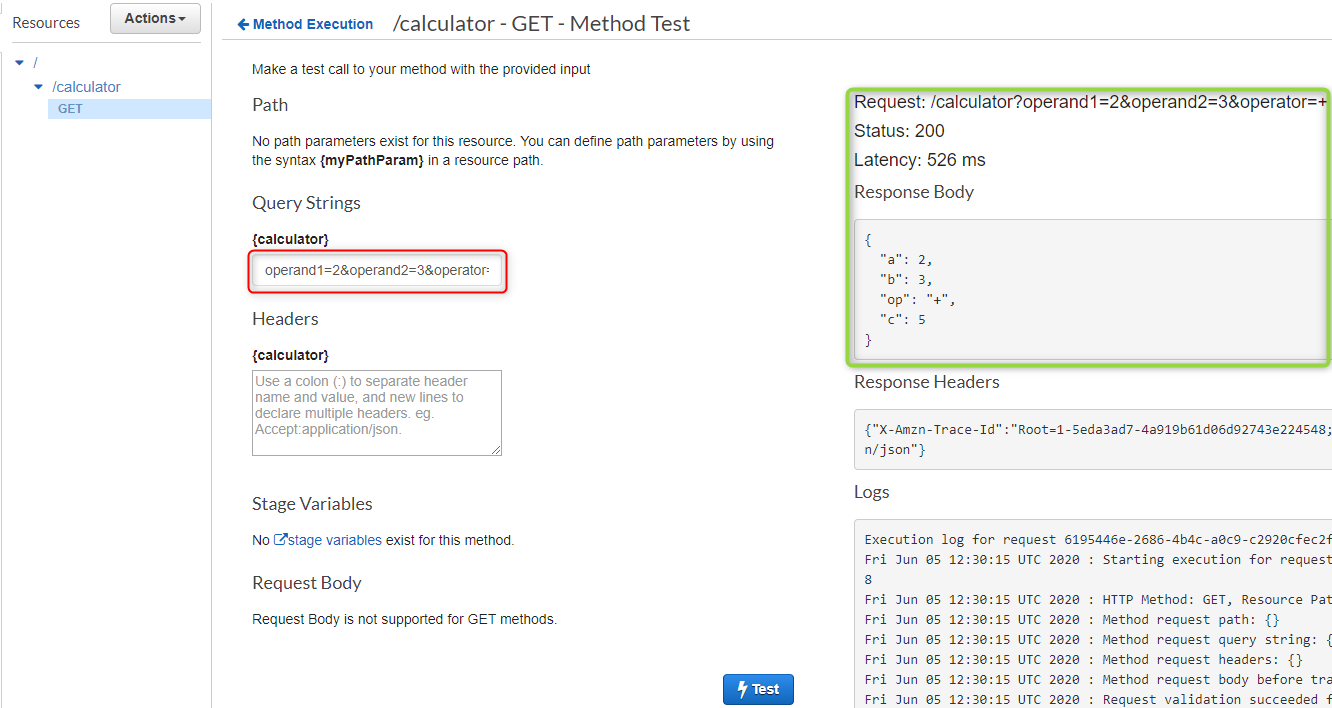
1. Choose **Method Execution**.
2. You can now test your GET method to verify that it has been properly set up to invoke the Lambda function.



* 1. For **Query Strings**, type **operand1=2&operand2=3&operator=+**
  2. Choose **Test**.



The results should look similar to this:



## Create the POST method with a JSON payload to call the Lambda function

After you complete the next step (Step 6), go back here and create the POST method. Consider this as an optional self-task. How is the POST method different from the GET method discussed above?

Refer to the guideline for more technical details: <https://docs.aws.amazon.com/apigateway/latest/developerguide/integrating-api-with-aws-services-lambda.html#api-as-lambda-proxy-expose-post-method-with-json-body-to-call-lambda-function>

## Create a GET method with path parameters to call the Lambda function

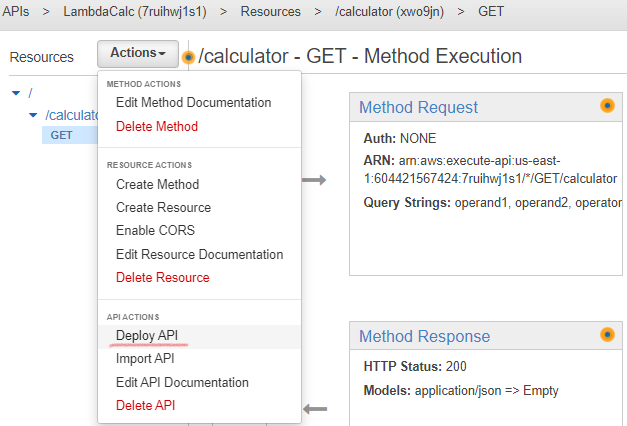
After you complete the next step (Step 6), go back here and create the GET method with path parameters. Consider this as an optional self-task. How is the GET method with path parameters different from the GET method with query parameters discussed above?

Refer to the guideline for more technical details: <https://docs.aws.amazon.com/apigateway/latest/developerguide/integrating-api-with-aws-services-lambda.html#api-as-lambda-proxy-expose-get-method-with-path-parameters-to-call-lambda-function>

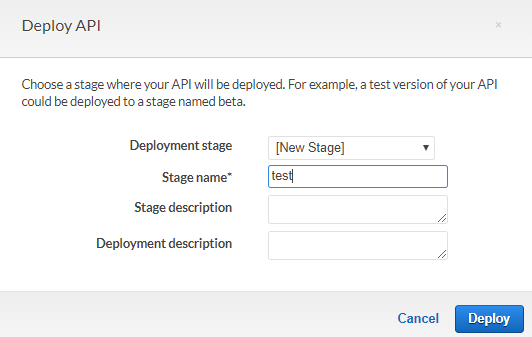
# Step 6: Deploying and testing the API

At this point the API can be called only via **Test Invoke** in the API Gateway console. To make it available to clients, you'll need to deploy it as follows:

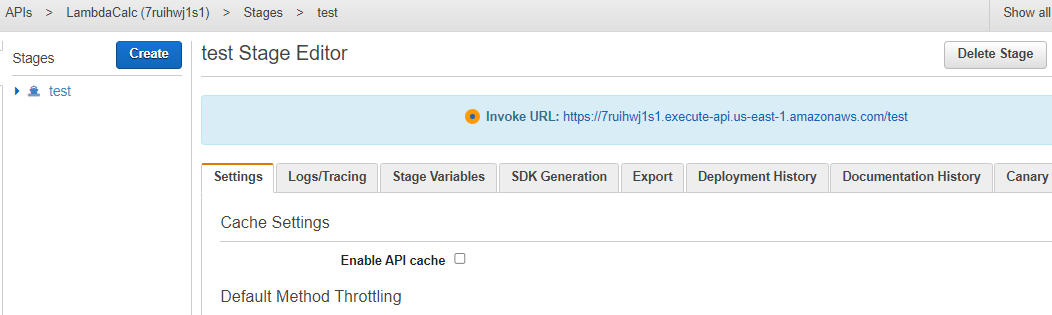
1. Choose **Deploy API** from the **Actions** dropdown menu.



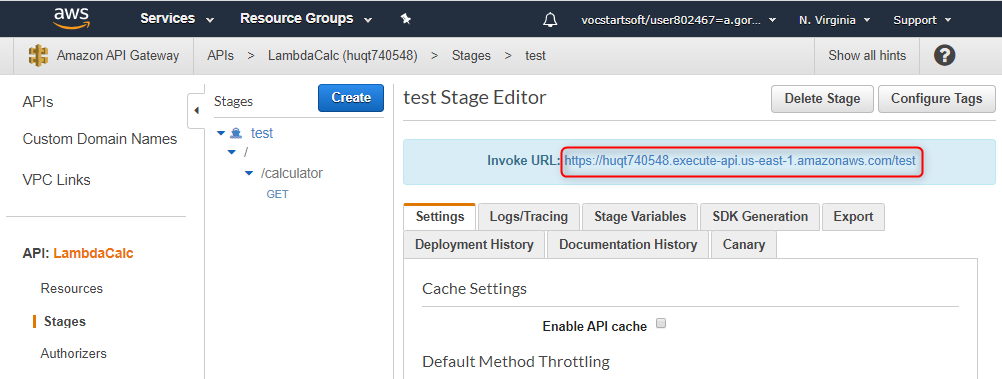
1. Choose **[New Stage]** from the **Deployment Stage** dropdown menu.
2. For **Stage Name**, enter **test**.



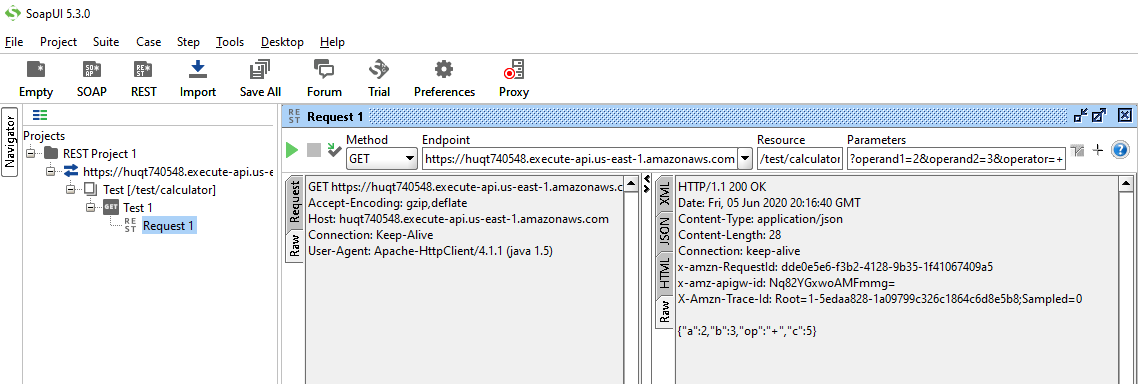
1. Choose **Deploy**.
2. Note the **Invoke URL** at the top of the console window.



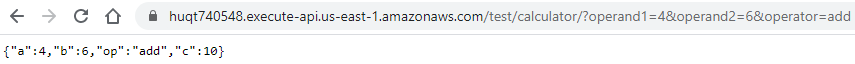
1. You need to add API name and parameter at the end of listed URL (e.g. **/calculator?operand1=2&operand2=3&operator=add**) to invoke API



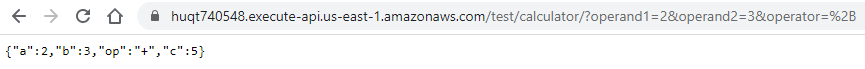
1. You can use this with tools such as [Postman](http://www.getpostman.com/), [cURL](https://curl.haxx.se/) or [SoapUI](https://www.soapui.org/downloads/soapui/) to test your API.



1. Test the API from a web browser



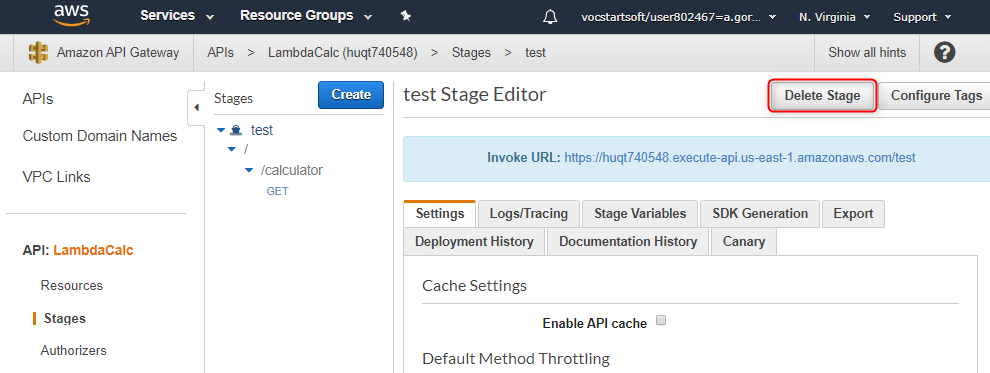
If you would like to use math symbols instead of the name of the operator, you should use code number, e.g. ‘+’ = %2B:



Always be sure to redeploy your API whenever you add, modify, or delete a resource or method, update a data mapping, or update stage settings. Otherwise, new features or updates will not be available to clients of your API.

# Step 7: Un-deploying the API and delete the Lambda function

*Delete the Stage to undeploy your API after completing the exercise!*



*Go to Lambda functions and delete Calculator function you created.*

