Deploy Serverless Applications with AWS Lambda and API Gateway

Reference this often? Create an account to bookmark tutorials.

10 MIN | PRODUCTS USED: Terraform

Serverless computing is a cloud computing model in which a cloud provider allocates compute resources on demand. This contrasts with traditional cloud computing where the user is responsible for directly managing virtual servers.

Most serverless applications use Functions as a Service (FaaS) to provide application logic, along with specialized services for additional capabilities such as routing HTTP requests, message queuing, and data storage.

In this tutorial, you will deploy a NodeJS function to AWS Lambda, and then expose that function

to the Internet using Amazon API Gateway.

Prerequisites

This tutorial assumes that you are familiar with the standard Terraform workflow. If you are new to Terraform, complete the Get Started tutorials first.

For this tutorial, you will need:

- The Terraform CLI (1.0.1+) installed.
- An AWS account.
- The AWS CLI (2.0+) installed, and configured for your AWS account.

Warning: Some of the infrastructure in this tutorial does not qualify for the AWS free tier. Destroy the infrastructure at the end of the guide to avoid unnecessary charges. We are not responsible for any charges that you incur.

Clone example configuration

Clone the Learn Terraform Lambda and API Gateway GitHub repository for this tutorial. If you're stuck at any point during this tutorial, refer

to the `final` branch to find the final configuration.

```
$ git clone https://github.com/hashicorp/l
```

Change to the repository directory.

```
$ cd learn-terraform-lambda-api-gateway
```

Review the configuration in `main.tf`. It defines the AWS provider you will use for this tutorial and an S3 bucket which will store your Lambda function.

Initialize this configuration.

```
$ terraform init
```

Apply the configuration to create your S3 bucket. Respond to the confirmation prompt with a `yes`.

```
$ terraform apply
## ...

Apply complete! Resources: 2 added, 0 chan

Outputs:

lambda_bucket_name = "learn-terraform-func
```

Create and upload Lambda function archive

To deploy an AWS Lambda function, you must package it in an archive containing the function source code and any dependencies.

The way you build the function code and dependencies will depend on the language and frameworks you choose. In this tutorial, you will deploy the NodeJS function defined in the Git repository you cloned earlier.

Review the function code in `hello-world/hello.js`.

hello-world/hello.js

```
module.exports.handler = async (event) => {
  console.log('Event: ', event);
  let responseMessage = 'Hello, World!';

  return {
    statusCode: 200,
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      message: responseMessage,
    }),
  }
}
```

This function takes an incoming event object from Lambda and logs it to the console. Then it returns an object which API Gateway will use to generate an HTTP response.

Add the following configuration to `main.tf` to package and copy this function to your S3 bucket.

main.tf

```
data "archive_file" "lambda_hello_world" {
  type = "zip"

  source_dir  = "${path.module}/hello-worl
  output_path = "${path.module}/hello-worl
}

resource "aws_s3_object" "lambda_hello_wor
  bucket = aws_s3_bucket.lambda_bucket.id

  key    = "hello-world.zip"
  source = data.archive_file.lambda_hello_

  etag = filemd5(data.archive_file.lambda_
}
```

This configuration uses the `archive_file` data source to generate a zip archive and an `aws_s3_object` resource to upload the archive to your S3 bucket.

Create the bucket object now. Respond to the confirmation prompt with a `yes`.

```
$ terraform apply
## ...

Apply complete! Resources: 1 added, 0 chan

Outputs:

lambda_bucket_name = "learn-terraform-func
```

Once Terraform deploys your function to S3, use the AWS CLI to inspect the contents of the S3 bucket.

```
$ aws s3 ls $(terraform output -raw lambda_
2021-07-08 13:49:46        353 hello-world
```

Create the Lambda function

Add the following to `main.tf` to define your Lambda function and related resources.

main.tf

```
resource "aws_lambda_function" "hello_worl
  function_name = "HelloWorld"

  s3_bucket = aws_s3_bucket.lambda_bucket.
  s3_key    = aws_s3_object.lambda_hello_w

  runtime = "nodejs12.x"
  handler = "hello.handler"

  source_code_hash = data.archive_file.lam

  role = aws_iam_role.lambda_exec.arn
}

resource "aws_cloudwatch_log_group" "hello_
  name = "/aws/lambda/${aws_lambda_functio
```

```
  retention_in_days = 30
}

resource "aws_iam_role" "lambda_exec" {
  name = "serverless_lambda"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Action = "sts:AssumeRole"
      Effect = "Allow"
      Sid    = ""
      Principal = {
        Service = "lambda.amazonaws.com"
      }
      }
    ]
  })
}

resource "aws_iam_role_policy_attachment"
  role       = aws_iam_role.lambda_exec.nai
  policy_arn = "arn:aws:iam::aws:policy/se
}
```

This configuration defines four resources:

- `aws_lambda_function.hello_world` configures the Lambda function to use the bucket object containing your function code. It also sets the runtime to NodeJS 12.x, and assigns the handler to the `handler` function defined in `hello.js`. The `source_code_hash` attribute will change

whenever you update the code contained in the archive, which lets Lambda know that there is a new version of your code available. Finally, the resource specifies a role which grants the function permission to access AWS services and resources in your account.

- `aws_cloudwatch_log_group.hello_world` defines a log group to store log messages from your Lambda function for 30 days. By convention, Lambda stores logs in a group with the name `/aws/lambda/<Function Name>`.

- `aws_iam_role.lambda_exec` defines an IAM role that allows Lambda to access resources in your AWS account.

- `aws_iam_role_policy_attachment.lambda_policy` attaches a policy the IAM role. The `AWSLambdaBasicExecutionRole` is an AWS managed policy that allows your Lambda function to write to CloudWatch logs.

Add the following to `outputs.tf` to create an output value for your Lambda function's name.

outputs.tf

```
output "function_name" {
  description = "Name of the Lambda functi

  value = aws_lambda_function.hello_world.
}
```

Apply this configuration to create your Lambda function and associated resources. Respond to the confirmation prompt with a `yes`.

```
$ terraform apply
## ...

Apply complete! Resources: 4 added, 0 chan

Outputs:

function_name = "HelloWorld"
lambda_bucket_name = "learn-terraform-func
```

Once Terraform creates the function, invoke it using the AWS CLI.

```
$ aws lambda invoke --region=us-east-1 --f
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

Check the contents of `response.json` to confirm that the function is working as expected.

```
$ cat response.json
{"statusCode":200,"headers":{"Content-Type
```

This response matches the object returned by the handler function in `hello-world/hello.js`.

You can review your function in the AWS Lambda Console.

Create an HTTP API with API Gateway

API Gateway is an AWS managed service that allows you to create and manage HTTP or WebSocket APIs. It supports integration with AWS Lambda functions, allowing you to implement an HTTP API using Lambda functions to handle and respond to HTTP requests.

Add the following to `main.tf` to configure an API Gateway.

main.tf

```
resource "aws_apigatewayv2_api" "lambda" {
  name          = "serverless_lambda_gw"
  protocol_type = "HTTP"
}

resource "aws_apigatewayv2_stage" "lambda"
  api_id = aws_apigatewayv2_api.lambda.id
```

```
  name        = "serverless_lambda_stage"
  auto_deploy = true

  access_log_settings {
    destination_arn = aws_cloudwatch_log_g

    format = jsonencode({
      requestId               = "$context.
      sourceIp                = "$context.
      requestTime             = "$context.
      protocol                = "$context.
      httpMethod              = "$context.
      resourcePath            = "$context.
      routeKey                = "$context.
      status                  = "$context.
      responseLength          = "$context.
      integrationErrorMessage = "$context.
      }
    )
  }
}

resource "aws_apigatewayv2_integration" "h
  api_id = aws_apigatewayv2_api.lambda.id

  integration_uri    = aws_lambda_function
  integration_type   = "AWS_PROXY"
  integration_method = "POST"
}

resource "aws_apigatewayv2_route" "hello_w
  api_id = aws_apigatewayv2_api.lambda.id

  route_key = "GET /hello"
  target    = "integrations/${aws_apigatew
}
```

```
resource "aws_cloudwatch_log_group" "api_gw
  name = "/aws/api_gw/${aws_apigatewayv2_a

  retention_in_days = 30
}

resource "aws_lambda_permission" "api_gw" {
  statement_id  = "AllowExecutionFromAPIGa
  action        = "lambda:InvokeFunction"
  function_name = aws_lambda_function.hell
  principal     = "apigateway.amazonaws.co

  source_arn = "${aws_apigatewayv2_api.lam
}
```

This configuration defines four API Gateway resources, and two supplemental resources:

- `aws_apigatewayv2_api.lambda` defines a name for the API Gateway and sets its protocol to `HTTP`.
- `aws_apigatewayv2_stage.lambda` sets up application stages for the API Gateway – such as "Test", "Staging", and "Production". The example configuration defines a single stage, with access logging enabled.
- `aws_apigatewayv2_integration.hello_world` configures the API Gateway to use your Lambda function.
- `aws_apigatewayv2_route.hello_world`

maps an HTTP request to a target, in this case your Lambda function. In the example configuration, the `route_key` matches any GET request matching the path `/hello`. A `target` matching `integrations/<ID>` maps to a Lambda integration with the given ID.

- `aws_cloudwatch_log_group.api_gw` defines a log group to store access logs for the `aws_apigatewayv2_stage.lambda` API Gateway stage.

- `aws_lambda_permission.api_gw` gives API Gateway permission to invoke your Lambda function.

The API Gateway stage will publish your API to a URL managed by AWS.

Add an output value for this URL to `outputs.tf`.

outputs.tf

```
output "base_url" {
  description = "Base URL for API Gateway s

  value = aws_apigatewayv2_stage.lambda.in
}
```

Apply your configuration to create the API Gateway and other resources. Respond to the

confirmation prompt with a `yes`.

```
$ terraform apply
## ...

Apply complete! Resources: 6 added, 0 chan

Outputs:

base_url = "https://mxg7cq38p4.execute-api
function_name = "HelloWorld"
lambda_bucket_name = "learn-terraform-func
```

Now, send a request to API Gateway to invoke the Lambda function. The endpoint consists of the `base_url` output value and the `/hello` path, which do you defined as the `route_key`.

```
$ curl "$(terraform output -raw base_url)/
{"message":"Hello, World!"}
```

Update your Lambda function

When you call Lambda functions via API Gateway's proxy integration, API Gateway passes the request information to your function via the `event` object. You can use information about the request in your function code.

Now, use an HTTP query parameter in your function.

In `hello-world/hello.js`, add an `if` statement to replace the `responseMessage` if the request includes a `Name` query parameter.

hello-world/hello.js

```
 module.exports.handler = async (event) =>
   console.log('Event: ', event)
   let responseMessage = 'Hello, World!';

+  if (event.queryStringParameters && even
+    responseMessage = 'Hello, ' + event.q
+  }
+
   return {
     statusCode: 200,
     headers: {
       'Content-Type': 'application/json',
     },
     body: JSON.stringify({
       message: responseMessage,
     }),
   }
 }
```

Apply this change now.

Since your source code changed, the computed `etag` and `source_code_hash` values have changed as well. Terraform will update your S3

bucket object and Lambda function.

Respond to the confirmation prompt with a `yes`.

```
$ terraform apply
## ...

Terraform will perform the following actio

  # aws_lambda_function.hello_world will b
  ~ resource "aws_lambda_function" "hello_
        id                             = "
      ~ last_modified                  = "
      ~ source_code_hash               = "
        tags                           = {
        # (18 unchanged attributes hidden)

        # (1 unchanged block hidden)
    }

  # aws_s3_object.lambda_hello_world will
  ~ resource "aws_s3_object" "lambda_hello_
      ~ etag               = "ba1ce6b2aa28
        id                 = "hello-world.
        tags               = {}
      + version_id         = (known after
        # (10 unchanged attributes hidden)
    }

Plan: 0 to add, 2 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions descr
  Only 'yes' will be accepted to approve.
```

```
  Enter a value: yes

aws_s3_object.lambda_hello_world: Modifyin
aws_s3_object.lambda_hello_world: Modifica
aws_lambda_function.hello_world: Modifying
aws_lambda_function.hello_world: Still mod
aws_lambda_function.hello_world: Modificat

Apply complete! Resources: 0 added, 2 chan

Outputs:

base_url = "https://iz85oarz9l.execute-api
function_name = "HelloWorld"
lambda_bucket_name = "learn-terraform-func
```

Now, send another request to your function, including the `Name` query parameter.

```
$ curl "$(terraform output -raw base_url)/
{"message":"Hello, Terraform!"}
```

Before cleaning up your infrastructure, you can visit the AWS Lambda Console for your function to review the infrastructure you created in this tutorial.

Clean up your infrastructure

You have created an AWS Lambda function with

an API Gateway integration. These components are essential parts of most serverless applications.

Before moving on, clean up the infrastructure you created by running the `terraform destroy` command.

```
$ terraform destroy
```

Remember to confirm your destroy with a `yes`.

Next steps

Now that you have learned how to deploy Lambda functions with Terraform, check out the following resources for more information.

- The Terraform Registry includes modules for Lambda and API Gateway, which support serverless development.
- Create and use Terraform modules to organize your configuration.
- Use the Cloud Development Kit (CDK) for Terraform to deploy multiple Lambda functions with TypeScript.

Was this tutorial helpful? Yes

No

Previous

Next