


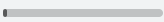


# AWS Lambda Environment Variables - Best Practices and Common Use Cases

 [blog.awsfundamentals.com/aws-lambda-environment-variables-best-practices-and-common-use-cases](https://blog.awsfundamentals.com/aws-lambda-environment-variables-best-practices-and-common-use-cases)

Tobias Schmidt



 Play this article

▶ 0:00 / 8:11   

At AWS Lambda, environment variables can be used to update a function's behavior without touching its code. This makes them a powerful asset for different use cases.

## Introduction

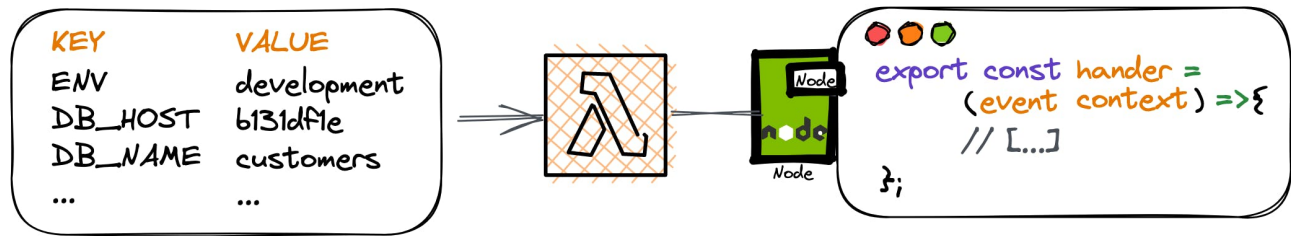
Before jumping into the practical aspects of working with environment variables at AWS Lambda, let's cover the fundamentals first.

## Key-Values Pairs Outside of Your Code

Environment variables are key-value pairs that can be used to store information outside of your actual code.

## How Environment Variables Are Used in AWS Lambda

At Lambda, environment variables are stored in the function's version-specific configuration.



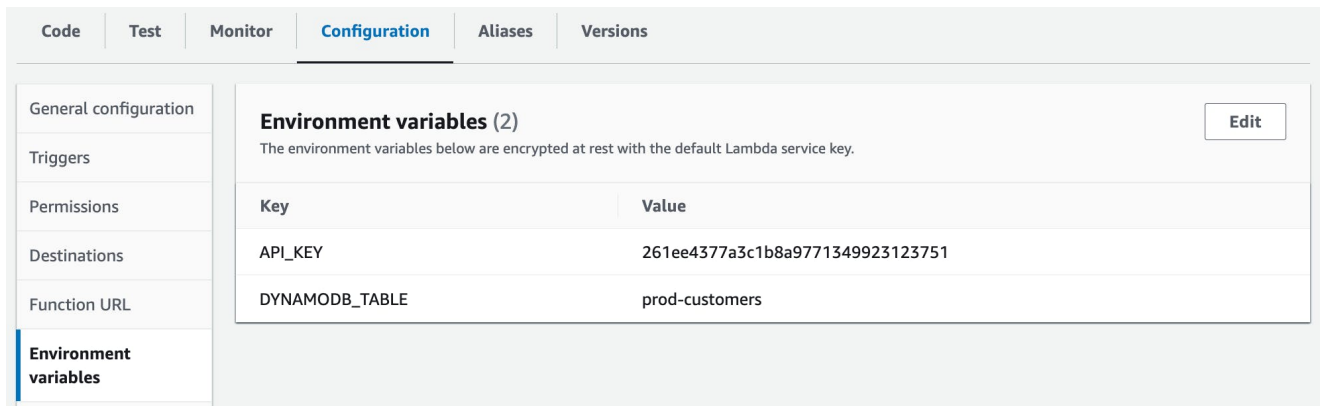
The Lambda runtime will make them available to your code when the function is initialized and also adds a set of default environment variables that you can work with.

## Working with Environment Variables in Lambda

You can set and update your function's environment variables in multiple ways. This includes manual ways via the AWS console or programmatical ones via the AWS CLI or Infrastructure-as-Code tools.

### Setting Environment Variables in the AWS Management Console

By clicking on the name of your function in the [AWS Lambda console](#) you'll be taken to its overview. Via a click on the **Configuration** tab below the visual representation of your Lambda function's integration, you'll notice that the **Environment Variables** link will appear in the new side navigation.

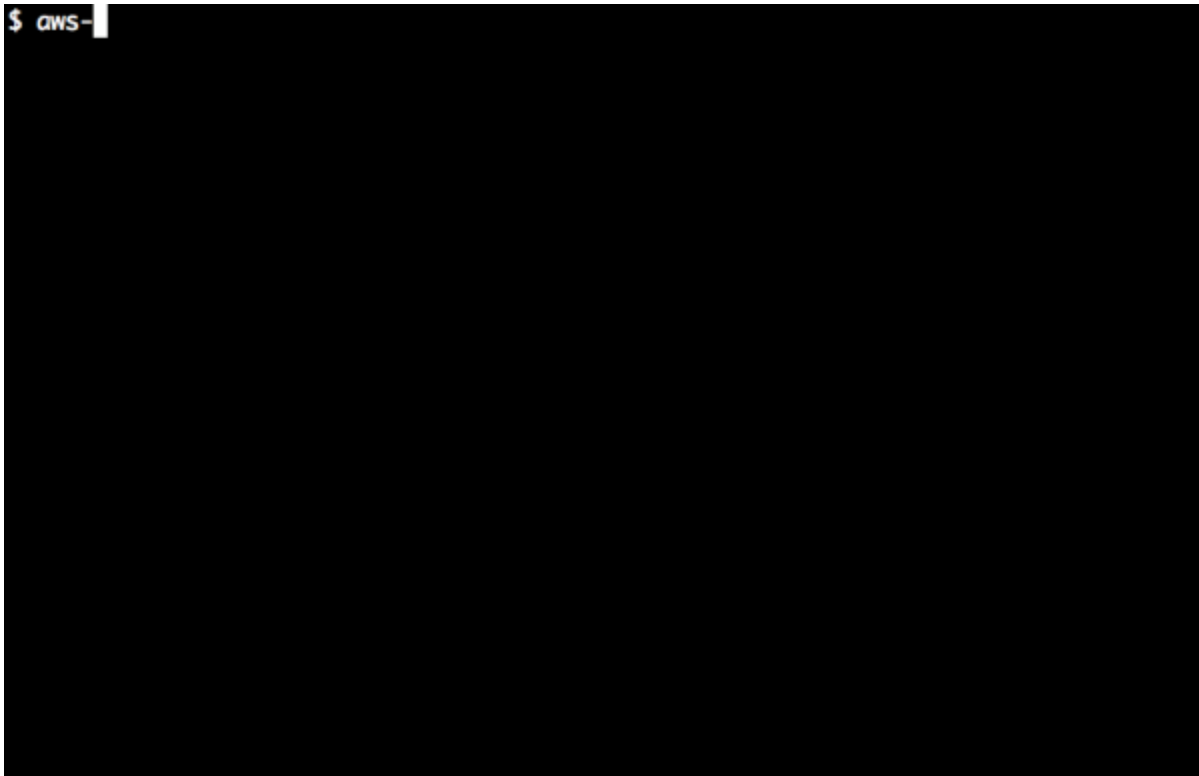


By clicking on edit you can add or remove environment variables.

### Setting Environment Variables through the AWS CLI

You're also able to update your function's environment variables programmatically in different ways. Each of them will work with the AWS API.

One of the simplest ways is to use the AWS CLI which you can install via many package managers, e.g. via [homebrew](#) for macOS and `brew install awscli`. If you don't like to jump to the documentation many times, have a look at [aws-shell](#) as it will give your CLI super-powers via auto-completion and integrated documentation.



Don't forget that you need to configure your credentials beforehand via `aws configure`. The necessary AWS Access Key ID and Secret Access Key pair can be created on your user's [security\\_credentials page](#).

When everything is set up properly (test with `aws sts get-caller-identity` - it has to return a valid response including your account ID), we can the current environment configuration for our function:

```
aws lambda get-function-configuration \
  --function-name myfunction \
  --region eu-central-1
```

This will print us everything related to our function configuration, including the memory configuration and VPC configuration. But we can also just return the environment variables directly by using a query.

```
aws lambda get-function-configuration \
  --function-name myfunction \
  --query 'Environment.Variables' \
  --region eu-central-1
```

We can also easily list all environment variables of all functions of a single region.

```
aws lambda list-functions \
  --query 'Functions[].{Name:FunctionName,Env:Environment.Variables}' \
  --output table --region eu-central-1
```

```
-----
|                               ListFunctions                               |
+-----+
|                               Name                                       |
+-----+
| myfunction                                                             |
+-----+
||                               Env                                       ||
+-----+
|||                               Variables                               ||| |
||+-----+-----+-----+-----+||
|||                               API_KEY                               |||
||+-----+-----+-----+-----+||
||| 261ee4377a3c1b8b2371349923123751 | customers                       |||
||+-----+-----+-----+-----+||
-----
```

Now that we've got the current configuration, we can easily update the environment variables via [update-function-configuration](#).

```
aws lambda update-function-configuration \
  --function-name my-function \
  --environment "Variables=ENV=dev,API_KEY=3b6f7fc24a9e3dbff9b" \
  --region eu-central-1
```

The response will include your whole configuration so you'll immediately see the updated environment variables.

## Setting Environment Variables through the AWS SDK

---

You can also make use of the AWS SDK to update the environment variables of a Lambda function using a script, e.g. with Node.js.

You can create a new mini-project from scratch via [npm init](#). Afterward, we need to install the AWS-SDK via [npm i aws-sdk](#) and we're good to go to adapt the [index.js](#) file:

```
const { Lambda } = require('aws-sdk');

const lambda = new Lambda({ region: 'eu-central-1' });

const update = async () => {
  await lambda.updateFunctionConfiguration(params)
    .promise()
    .catch(e => {
      console.error(`Failed to update env vars: ${err}`);
    });
};

update();
```

We can execute the script via `node index.js`.

## Setting Environment Variables through Infrastructure-As-Code Tools

---

Using the AWS CLI or the AWS-SDK is a convenient way to update our environment variables to quickly test something. But for a real-world application, this is not a proper way. We should rely on Infrastructure-as-Code (IaC) tools that enable us to define our whole infrastructure setup in our code base.

One of the great tools out there that are perfect for Lambda-powered applications is the Serverless Framework. It abstracts away a lot of tedious work by putting everything into a rather simple YAML template that will later be internally translated into a proper CloudFormation template.

You can quickly install it via `npm i -g serverless` and you're already good to go.

By using `sls` we can initialize a project from a basic template, e.g. `AWS - Node.js - HTTP API`. This will result in a small handler function file (`handler.js`) and the Serverless infrastructure definition file (`serverless.yml`).

In our YAML file, we can quickly add environment variables:

```
service: aws-node-http-api-project
frameworkVersion: '2 || 3'

provider:
  name: aws
  runtime: nodejs12.x
  lambdaHashingVersion: '20201221'

functions:
  hello:
    handler: handler.hello
    environment:
      - ENV: dev
      - DYNAMODB_TABLE: customers
    events:
      - httpApi:
          path: /
          method: get
```

That's already it. With `sls deploy` we can create our stack including the function and even an AWS API Gateway! We can also simply delete our stack with a single command by running `sls remove`.

## Tips and Best Practices for Using Environment Variables in AWS Lambda

---

Using environment variables with AWS Lambda is easy to start with. Nevertheless, there are a few things you should remember and some best practices you should rely on.

### Reserved Environment Variables by AWS Lambda

---

There are reserved environment variables that are filled at the initialization of your Lambda function.

This includes amongst others:

- `AWS_REGION` – Your functions AWS region, e.g. `us-east-1`.
- `AWS_EXECUTION_ENV` – The configured runtime, e.g. `AWS_Lambda_nodejs18.x`.
- `AWS_LAMBDA_FUNCTION_NAME` – The name of your function.
- `AWS_LAMBDA_FUNCTION_MEMORY_SIZE` – The configured memory.
- `AWS_LAMBDA_FUNCTION_VERSION` – The version of the executed function.

You can find the complete list of defined runtime environment variables for AWS Lambda in the [AWS documentation](#).

## Reusable and Consistent Configuration across Functions

---

With Infrastructure-as-Code tools, it doesn't require much effort to provision many AWS Lambda functions.

Often, your different functions rely on the same configuration values, e.g. the DynamoDB table they have to access. Generally, this should not be hardcoded in your function's code, but abstracted to environment variables.

And this makes environment variables in combination with IaC tools beautiful, as you can spread environment variables that are needed by multiple functions very easily.

Let's extend our Serverless Framework example project from above, by putting one environment variable into a global template variable and creating another function.

```
service: aws-node-http-api-project
frameworkVersion: '2 || 3'

custom:
  dynamoDbTable: customers

provider:
  name: aws
  runtime: nodejs12.x
  lambdaHashingVersion: '20201221'

functions:
  hello:
    handler: handler.hello
    environment:
      - ENV: dev
      - DYNAMODB_TABLE: ${self:custom.dynamoDbTable}
  second:
    handler: another-handler.hello
    environment:
      - ENV: dev
      - DYNAMODB_TABLE: ${self:custom.dynamoDbTable}
```

And there we go: our environment variable is defined in a single place, but spread across multiple functions. We don't need to worry about updating it only in one place, but forgetting about another.

## Use of Environment Variables for Configuration Values

---

With environment variables, it's easy to keep your function's code very generic so it doesn't include any specifics for your stage.

This means you can deploy the same function code to the development and production stage but only adapt the environment variables to configure everything properly.

## **Encrypting Environment Variables to Store Sensitive Information**

---

Environment variables are often used for storing sensitive information carelessly. This is not a good practice, as those values are available to everybody that has enough permissions to access the function's configuration.

But AWS Lambda allows you to encrypt the environment variables so that there's another layer of protection at transit. This won't allow accessing the environment variables via the console or API, even if the user has enough permissions.

The values can be decrypted with the corresponding KMS key in your Lambda function's code. You can read the full guide in the [AWS documentation](#).

Generally, it's recommended to use the AWS Secrets Manager to store sensitive information.

## **Restricting Access to Environment Variables through IAM Policies**

---

It's also possible to restrict IAM users from accessing Lambda's environment variables through the console or the AWS API.



```
{
  "Id": "MyCustomKey",
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Deny Access to Environment Variables",
      "Effect": "Deny",
      "Principal": {
        "AWS": [
          "arn:aws:iam::123456789012:another-user"
        ]
      },
      "Action": [
        "kms:Encrypt",
        "kms:Decrypt",
        "kms:ReEncrypt*",
        "kms:GenerateDataKey*",
        "kms:DescribeKey"
      ],
      "Resource": "*"
    }
  ]
}
```

**Be aware:** this doesn't protect that variables are logged to the console so they end up in CloudWatch.

## Conclusion

---

Environment variables in AWS Lambda are a powerful tool for customizing the behavior of a function without changing its code.

These key-value pairs can be set and updated in several ways. The AWS CLI and AWS SDK offer the ability to update variables programmatically, and Infrastructure-as-Code tools provide a more robust and scalable solution for real-world applications.

Regardless of the method chosen, environment variables provide a flexible and convenient way to modify the behavior of AWS Lambda functions.

## Published on

---