HOW TO RUN ANY PROGRAMMING LANGUAGE ON AWS LAMBDA: CUSTOM RUNTIMES.

AWS Lambda | C++ | Lambda Custom Runtime | Linux Bash | Serverless

beSharp | 10 August 2020

AWS Lambda Functions as a Service (FaaS) have quickly become the Swiss Army knife of the AWS cloud DevOps since it is possible to use them for a plethora of different tasks. From the backend of a web application to the ingestion of an AWS IoT application, from simple infrastructure automation to the real-time analysis of messages delivered through AWS SQS or AWS Kinesis. Furthermore, they are cheap, versatile, scalable, reasonably priced, and trivial to set up and maintain.

To make them more attractive for developers and devops AWS offers an increasing number of Lambda runtimes, which allow you to write your code in different versions of several programming languages. At the moment of this writing, AWS Lambda natively supports Java, Go, PowerShell, Node.js, C#, Python, and Ruby.

However, there are lots of other programming languages out there that you may want to use in your Lambda function to migrate to AWS Lambda existing applications currently deployed on-premises or on EC2 instances. Since rewriting a current code is often infeasible due to lack of time or missing libraries and functionality AWS recently began to offer a new possibility: **you can create a custom Runtime for Lambda and use the programming language you prefer!**

A Lambda function with a custom runtime environment differs from a normal lambda function because it not only contains the code the Lambda will run when the function is invoked but also all the compiled libraries needed to make the code run and – if the language you choose is interpreted like PHP or just in time compiled like Julia – you also need to include the binary interpreter. For most programming languages, a custom runtime prepared by third parties is usually available on github and is often either usable directly or a good base for a custom solution.

In the following section, however, we will describe how to create a generic runtime and present two examples created by AWS: **bash and C++**. Finally, we will compare the lambda response time of the custom runtimes with the one of a native runtime (Python 3.8). Creating a custom runtime also

allows us to understand how the lambda service works under the hood, which is often very useful for solving problems that may arise with lambda functions (e.g., cold starts).

How does Lambda Works?

AWS Lambda consists of two main parts: the **Lambda *service*** which manages the execution requests, and the **Amazon Linux micro virtual machines** provisioned using AWS Firecracker, which actually runs the code. A Firecracker VM is started the first time a given Lambda function receives an execution request (the so-called "Cold Start"), and as soon as the VM starts, it begins to poll the Lambda service for messages. When the VM receives a message, it runs your function code handler, passing the received message JSON to the function as the event object.

Thus every time the Lambda service receives a Lambda execution request, it checks if there is a Firecracker microVM available to manage the execution request. If so, it delivers the message to the VM to be executed. In contrast, if no available Firecracker VM is found, it starts a new VM to manage the message.

Each VM executes one message at a time, so if a lot of concurrent requests are sent to the Lambda service, for example due to a traffic spike received by an API gateway, several new Firecracker VMs will be started to manage the requests and the average latency of the requests will be higher since each VM takes roughly a second to start.

In a lambda function using a native runtime, you do not need to worry about how your function will poll for messages from the lambda service and send execution reports back to the lambda service; the native runtime will take care of that for you. However, this is not the case for a custom runtime. When a Lambda function is created with a custom runtime, AWS Lambda Service will start a basic AmazonLinux VM without any installed libraries except for vanilla bash and a few basic unix commands (e.g., ls, curl). Differently from a normal Lambda, in addition to your code and external libraries, you'll also need to include in the deployment package a script or executable called "bootstrap" that will manage the interaction between the function VM and the Lambda service. AWS Lambda Service exposes a simple HTTP interface for runtimes to receive invocation events from Lambda and send response data back.

The bootstrap program needs to perform the following tasks:

1. **Get an event**: Call the invocation API to get the next event. The response body contains the event data. Response headers contain the request ID and other information.

1. **Propagate:** the tracing header – Get the X-Ray tracing header from the Lambda-Runtime-Trace-Id header in the API response. Set the _X_AMZN_TRACE_ID environment variable locally with the same value. The X-Ray SDK uses this value to connect trace data between services.

1. **Create a context object**: Create an object with context information from environment variables and headers in the API response.

1. **Invoke the function handler**: Pass the event and context object to the handler.

1. **Handle the response**: Call the invocation response API to post the response from the handler.

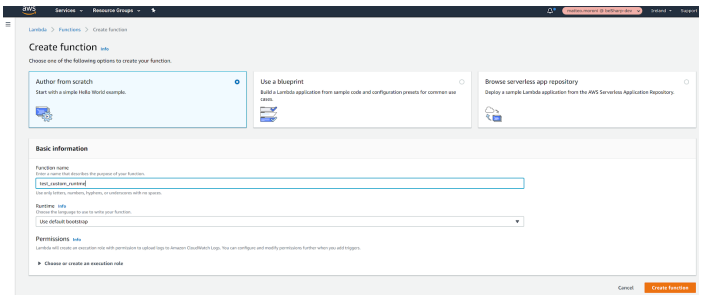
1. **Handle errors**: If an error occurs, call the invocation error API.

1. **Cleanup**: Release unused resources, send data to other services, or perform additional tasks before getting the next event.

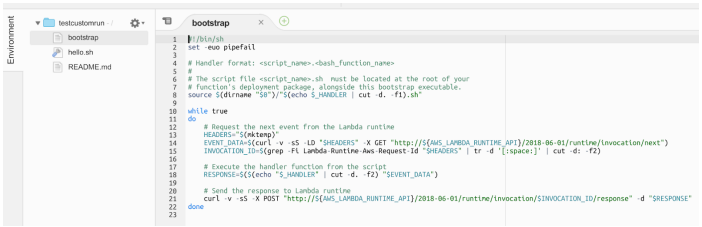
The bootstrap script/executable and other language level libraries and interpreters (e.g., PHP interpreter) can be included in a dedicated lambda layer to generate a generic and portable custom runtime used on several Lambda Functions.

How to create a Bash Lambda with custom runtimes

The easiest way to get going with custom runtime is through the AWS Console: from the Lambda Service Dashboard select Create Lambda and in the runtime section select Custom Runtime with Use Default bootstrap and click Create Function



Using these default settings, the Lamba service will create a basic Bash Lambda with a default bootstrap script. Let's take a look at the previously generated bootstrap script:




```
#!/bin/sh
set -euo pipefail

# Handler format: .
#
# The script file .sh  must be located at the root of your
# function's deployment package, alongside this bootstrap executable.
source $(dirname "$0")/"$(echo $_HANDLER | cut -d. -f1).sh"

while true
do
    # Request the next event from the Lambda runtime
    HEADERS="$(mktemp)"
    EVENT_DATA=$(curl -v -sS -LD "$HEADERS" -X GET "http://${AWS_LAMBDA_RUNTIME_API}/
2018-06-01/runtime/invocation/next")
    INVOCATION_ID=$(grep -Fi Lambda-Runtime-Aws-Request-Id "$HEADERS" | tr -d '[:spac
```

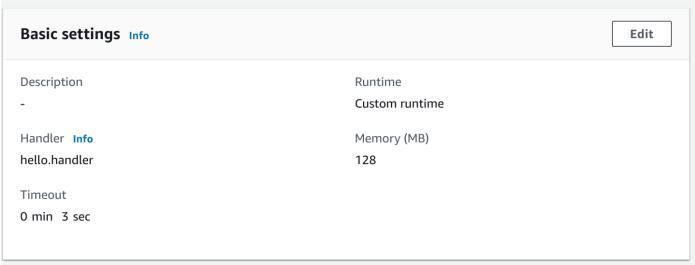
```
e:]' | cut -d: -f2)

    # Execute the handler function from the script
    RESPONSE=$($(echo "$_HANDLER" | cut -d. -f2) "$EVENT_DATA")

    # Send the response to Lambda runtime
    curl -v -sS -X POST "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$INVOCATION_ID/response" -d "$RESPONSE"
done
```

Let's quickly review what this script does:

- the first line set -euo pipefail just makes sure that the script is terminated immediately if an exception or an empty variable is encountered.
- The _HANDLER environment variable is valorized during the start of the VM and contains the file and function name of our lambda handler in the format <script_name>.<bash_function_name>



- source $(dirname "$0")/"$(echo $_HANDLER | cut -d. -f1).sh" just loads in memory the variables and function in the lambda handler file (hello.sh in our case)
- An infinite while loop is than started where first of all an Api call is dispatch to the lambda service endpoint (http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next) in order to get the next event to process. It should be noted that the endpoint is dynamic and is valorized in the AWS_LAMBDA_RUNTIME_API variable.
- The response is then evaluated by forwarding the event to the handler function (RESPONSE=$($(echo "$_HANDLER" | cut -d. -f2) "$EVENT_DATA"))
- Finally the result of the execution is returned to the Lambda Service through another Api call (curl -v -sS -X POST "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$INVOCATION_ID/response" -d "$RESPONSE")

It should be noted that in a real complete custom runtime you'll also need to manage errors and exceptions by calling the error invocation by calling the Invocation error API (/runtime/invocation/AwsRequestId/error) when an exception is raised by the Handler method:

```
REQUEST_ID=156cb537-e2d4-11e8-9b34-d36013741fb9
ERROR="{\"errorMessage\" : \"Error parsing event data.\", \"errorType\" : \"InvalidEventDataException\"}"
curl -X POST "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/error" -d "$ERROR" --header "Lambda-Runtime-Function-Error-Type: Unhandled"
```

A simple c++ example: Calculating the first n digits of pi in Lambda

Let's now move to a more complicated example: calculating the first n digits of pi in Lambda using a trivial and inefficient version of the Spigot Algorithm.

Running complicated calculations in Lambda function is often non-trivial both due to the lacking computational power reserved to the Firecracker VM (at least the low memory ones, memory and CPU scales proportionally in lambda function) and to the nature of the languages of native runtimes which are not too suited to high-performance computations (except Go). Conversely, C++ has a long and successful history in high-performance computing with lots of libraries available, from arbitrary precision Arithmetic to Matrix computations, from Fluid Dynamics to Particle Collisions.

Furthermore, this language is a first-class citizen in AWS with a full AWS SDK and a Lambda Runtimes builder developed and maintained directly by AWS.

To create our example, we can just clone the AWS git repository of the Lambda runtime builder and build the library using the commands (on UNIX):

```
$ git clone https://github.com/awslabs/aws-lambda-cpp.git
$ cd aws-lambda-cpp
$ mkdir build
$ cd build
$ cmake .. -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=~/lambda-install
$ make && make install
```

After that let's move to the Api Gateway Example in the examples folder and change the code in the main.cpp with:

```
#include <aws/lambda-runtime/runtime.h>
#include <aws/core/utils/json/JsonSerializer.h>
#include <aws/core/utils/memory/stl/SimpleStringStream.h>
#include

using namespace aws::lambda_runtime;

void pi_digits(int x, Aws::SimpleStringStream &s)
{
    long x_initial = x;
    unsigned long long k = 2;
    unsigned long long a = 4;
    unsigned long long b = 1;
    unsigned long long a1 = 12;
    unsigned long long b1 = 4;
    while (x > 0) {
        unsigned long long p = k * k;
        unsigned long long q = 2 * k +1;
        k = k + 1;

        unsigned long long a1old = a1;
        unsigned long long b1old = b1;
```

```
        a1 = p * a + q * a1;
        b1 = p * b + q * b1;
        a = a1old;
        b = b1old;

        long double d = a / b;
        long double d1 = a1 / b1;

        while ((d == d1) && (x > 0)) {
            s << static_cast(floor(d));
            if (x_initial == x) {
                s << ".";
            }
            x -= 1;
            a = 10 * (a % b);
            a1 = 10 * (a1 % b1);
            d = a / b;
            d1 = a1 / b1;
        }
    }
}


invocation_response my_handler(invocation_request const& request)
{
    using namespace Aws::Utils::Json;

    JsonValue json(request.payload);
    if (!json.WasParseSuccessful()) {
        return invocation_response::failure("Failed to parse input JSON", "InvalidJSO
N");
    }

    auto v = json.View();
    Aws::SimpleStringStream ss;
//
//    pi_digits(10, ss);
    if (v.ValueExists("queryStringParameters")) {
        auto query_params = v.GetObject("queryStringParameters");
        pi_digits((query_params.ValueExists("number") && query_params.GetObject("numb
er").IsString() ? stol(query_params.GetString("number")) : 10), ss);
    }

    JsonValue resp;
    resp.WithString("message", ss.str());

    return invocation_response::success(resp.View().WriteCompact(),
"application/json");
}

int main()
{
    run_handler(my_handler);
    return 0;
}
```

The Spigot algorithm used here is a C++ version of the one proposed here.

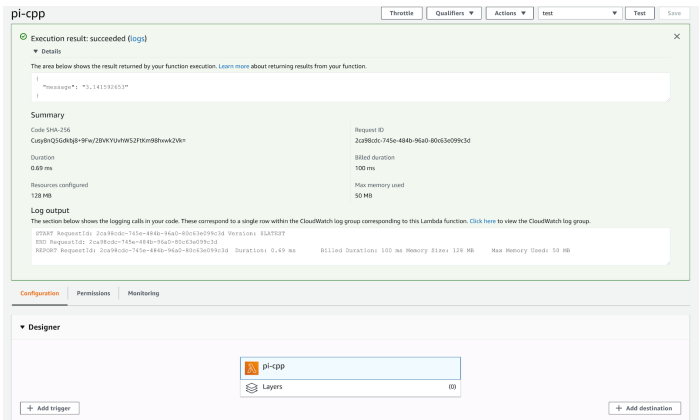
At this point we need to build the core of the AWS SDK C++ library to get the fancy JSON parsing tools you can see in the code above. To do so, I strongly advise you to use this command (tested on linux):

```
$ mkdir ~/install
$ git clone https://github.com/aws/aws-sdk-cpp.git
$ cd aws-sdk-cpp
$ mkdir build
$ cd build
$ cmake .. -DBUILD_ONLY="core" \
  -DCMAKE_BUILD_TYPE=Release \
  -DBUILD_SHARED_LIBS=OFF \
  -DENABLE_UNITY_BUILD=ON \
  -DCMAKE_INSTALL_PREFIX=~/install \
  -DENABLE_UNITY_BUILD=ON
$ make
$ make install
```

Now you can go back to the folder of the example and build your lambda app using:

```
$ mkdir build
$ cd build
$ cmake .. -DCMAKE_BUILD_TYPE=Release -DCMAKE_PREFIX_PATH=~/install
$ make
$ make aws-lambda-package-api
```

At this point, you can finally deploy the zip file generated by the build command on AWS Lambda to finally obtain your API gateway compatible pi digits computing Lambda which runs in less than 1 ms.



The same code executed in Python takes nearly four times as much to complete!

Finally, connecting the Lambda to API Gateway, you can obtain an endpoint able to calculate pi backed by C++, and the number of digits can be specified as query params.

As a note, this toy script can only calculate digits of pi up to 10 before the overflow of the unsigned long long counters. Improving the code to use GMP for arbitrary arithmetics to obtain Pi with an arbitrary number of digits is left as an exercise for the reader.

To conclude in this article, we explained how Lambda Custom runtimes work and presented two simple examples in bash and C++. Using custom runtimes adds many possible usages for the already extremely useful AWS Lambda service adding the possibility to run quick computation using high-performance Languages like C, C++, Rust, and Julia. Furthermore, custom runtimes also allow one to use Lambda to run simple bash scripts or even to migrate to the serverless paradigm existing PHP Apis.

If you are interested in this topic, do not hesitate to contact us.

beSharp

Dal 2011 beSharp guida le aziende italiane sul Cloud. Dalla piccola impresa alla grande multinazionale, dal manifatturiero al terziario avanzato, aiutiamo le realtà più all'avanguardia a realizzare progetti innovativi in campo IT.

Get in touch

beSharp.it
proud2becloud@besharp.it