**Learning Rust: Hello AWS**

[[Adrian Macal](https://amacal.medium.com/?source=post_page-----6775ee1236e3--------------------------------)](https://amacal.medium.com/?source=post_page-----6775ee1236e3--------------------------------)

[[Towards Dev](https://towardsdev.com/?source=post_page-----6775ee1236e3--------------------------------)](https://towardsdev.com/?source=post_page-----6775ee1236e3--------------------------------)

[Adrian Macal](https://amacal.medium.com/?source=post_page-----6775ee1236e3--------------------------------)

·

Follow

Published in

[Towards Dev](https://towardsdev.com/?source=post_page-----6775ee1236e3--------------------------------)

·

9 min read

·

3 days ago

33

2



Did you know that Rust is ready for AWS Lambda? Let’s explore together how it could accelerate your next game-changing project.

In this tutorial I am going to start small with very naive function and end up building fully functional HTTP server. I will cover local development and deployment using Terraform.

Let’s manually create the skeleton of our first Lambda function. The code is incredibly simple and echoes JSON input in some way.

use lambda\_runtime::{service\_fn, LambdaEvent, Error};  
use serde\_json::{json, Value};  
  
async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {  
 Ok(json!({ "hello": event.payload }))  
}  
  
#[tokio::main]  
async fn main() -> Result<(), Error> {  
 lambda\_runtime::run(service\_fn(handler)).await  
}

To build it successfully, we need to add a few crates that connect all the dots. You are probably familiar with tokio and serde\_json. The code requires one additional crate lambda\_runtime to emulate Lambda environment.

[package]  
name = "metrics-mingle-api"  
version = "0.1.0"  
edition = "2021"  
  
[[bin]]  
name = "metrics-mingle-api"  
path = "main.rs"  
  
[dependencies]  
lambda\_runtime = { version = "0.9.0" }  
serde\_json = { version = "1.0.108" }  
tokio = { version = "1.35.1" }

Now, we can successfully execute cargo build. Unfortunately, cargo run won’t work. Yes, we’ve built an executable, but it’s designed to run within AWS. How can we test it locally? We need to install the cargo-lambda tool to host a local version of our little AWS environment. Since I am in a VSCode DevContainer, I can easily do this by modifying my Dockerfile:

FROM mcr.microsoft.com/devcontainers/rust:bookworm  
RUN apt update && apt install -y python3-pip  
RUN pip3 install --break-system-packages cargo-lambda  
  
USER vscode  
RUN rustup update 1.75.0  
RUN rustup default 1.75.0

It’s time to execute our first request. In the background, we need to run a server on localhost:9000, which will receive all our requests and invoke the function with a JSON payload. Finally, the function returns the transformed output.

# run the server   
vscode ➜ /src $ cargo lambda watch -a 127.0.0.1  
 INFO invoke server listening on 127.0.0.1:9000  
 INFO starting lambda function function="\_" manifest="Cargo.toml"  
 Finished dev [unoptimized + debuginfo] target(s) in 0.05s  
 Running `/tmp/cargo/debug/metrics-mingle-api`  
  
# invoke the function  
vscode ➜ /src $ cargo lambda invoke -a 127.0.0.1 --data-ascii '"AWS"'  
{"hello":"AWS"}

Let’s jump into some infrastructure topics. AWS recommends Lambda deployment using ZIP files either directly from the local filesystem or S3 in all its tutorials. However, we’re going to opt for a more predictable method using ECR, which requires some DevOps skills. Let’s construct a Dockerfile to prepare the final image, ready for deployment to ECR:

FROM rust:1.75.0-bookworm as builder  
ENV CARGO\_TARGET\_DIR=/tmp/cargo  
  
RUN apt update && apt install -y python3-pip  
RUN pip3 install --break-system-packages cargo-lambda  
  
WORKDIR /build  
ADD src src  
RUN cd src && cargo lambda build --release  
  
FROM public.ecr.aws/lambda/provided:al2023  
COPY --from=builder /tmp/cargo/lambda/metrics-mingle/bootstrap ${LAMBDA\_RUNTIME\_DIR}  
CMD ["rust.handler"]

This Dockerfile mirrors our previous actions to configure VSCode and compile our Lambda manually. Once the image is built, we can launch a new container to test if it works. It’s quite straightforward in BASH:

# start new server  
vscode ➜ /src $ docker run -itp 9000:8080 metrics-mingle-api  
30 Dec 2023 13:19:43,582 [INFO] (rapid) exec '/var/runtime/bootstrap' (cwd=/var/task, handler=)  
START RequestId: 392dd9ce-3690-4bc2-b19f-f4494cdf0dd5 Version: $LATEST  
30 Dec 2023 13:20:16,647 [INFO] (rapid) INIT START(type: on-demand, phase: init)  
30 Dec 2023 13:20:16,653 [INFO] (rapid) INIT RTDONE(status: success)  
30 Dec 2023 13:20:16,653 [INFO] (rapid) INIT REPORT(durationMs: 5.822000)  
30 Dec 2023 13:20:16,653 [INFO] (rapid) INVOKE START(requestId: ebb243f2-0265-404c-8254-6586c3ccbc2e)  
30 Dec 2023 13:20:16,654 [INFO] (rapid) INVOKE RTDONE(status: success, produced bytes: 0, duration: 0.961000ms)  
END RequestId: ebb243f2-0265-404c-8254-6586c3ccbc2e  
REPORT RequestId: ebb243f2-0265-404c-8254-6586c3ccbc2e Init Duration: 0.04 ms Duration: 7.64 ms Billed Duration: 8 ms Memory Size: 3008 MB Max Memory Used: 3008 MB  
  
# invoke the function  
vscode ➜ /src $ curl --silent -XPOST "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{"payload":"hello world!"}'  
{"hello":{"payload":"hello world!"}}

The output already resembles a production environment. Let’s now transition from the local setup to actual deployment. We’ll begin by using Terraform to establish some basic infrastructure for ECR:

resource "aws\_ecr\_repository" "api" {  
 name = "metrics-mingle-api"  
 image\_tag\_mutability = "IMMUTABLE"  
  
 image\_scanning\_configuration {  
 scan\_on\_push = false  
 }  
}

After setting this up, we can manually push the image we previously built. This process involves three steps:

# build new image  
docker build -t metrics-mingle-api .  
  
# tag it using current timestamp  
docker tag metrics-mingle-api:latest 868404297900.dkr.ecr.eu-west-1.amazonaws.com/metrics-mingle-api:202312301441  
  
# push it to ECR  
docker push 868404297900.dkr.ecr.eu-west-1.amazonaws.com/metrics-mingle-api:202312301441

With the image now in ECR, we can provision the Lambda by pointing to the newly deployed image and granting it minimal permissions for logging to CloudWatch.

data "aws\_ecr\_image" "api" {  
 repository\_name = "metrics-mingle-api"  
 most\_recent = true  
}  
  
resource "aws\_lambda\_function" "api" {  
 function\_name = "metrics-mingle-api"  
 package\_type = "Image"  
 memory\_size = 128  
 role = aws\_iam\_role.api\_lambda.arn  
 image\_uri = "868404297900.dkr.ecr.eu-west-1.amazonaws.com/metrics-mingle-api@${data.aws\_ecr\_image.api.image\_digest}"  
  
 lifecycle {  
 ignore\_changes = [ image\_uri ]  
 }  
}  
  
resource "aws\_iam\_role" "api\_lambda" {  
 name = "metrics-mingle-api-lambda"  
  
 assume\_role\_policy = jsonencode({  
 Version = "2012-10-17",  
 Statement = [  
 {  
 Action = "sts:AssumeRole",  
 Effect = "Allow",  
 Principal = {  
 Service = "lambda.amazonaws.com"  
 }  
 },  
 ]  
 })  
}  
  
resource "aws\_iam\_role\_policy" "api\_lambda" {  
 role = aws\_iam\_role.api\_lambda.id  
 name = "metrics-mingle-api-lambda"  
  
 policy = jsonencode({  
 Version = "2012-10-17",  
 Statement = [  
 {  
 Effect = "Allow",  
 Action = [  
 "logs:CreateLogStream",  
 "logs:PutLogEvents"  
 ],  
 Resource = ["\*"]  
 }  
 ]  
 })  
}  
  
resource "aws\_cloudwatch\_log\_group" "api\_lambda" {  
 name = "/aws/lambda/metrics-mingle-api"  
 retention\_in\_days = 7  
}

Let’s test it. This time, I used the AWS UI to trigger the Lambda function twice with the default payload. It worked as expected. How fast was it? In both instances, it required less than 2ms, with the cold start taking around 200ms, which is acceptable.

START RequestId: 4f752192-b0bb-408a-ab19-13d5808f1fc2 Version: $LATEST  
END RequestId: 4f752192-b0bb-408a-ab19-13d5808f1fc2  
REPORT RequestId: 4f752192-b0bb-408a-ab19-13d5808f1fc2 Duration: 1.74 ms Billed Duration: 96 ms Memory Size: 128 MB Max Memory Used: 15 MB Init Duration: 93.36 ms   
START RequestId: 987264ab-9484-4dcd-bc6d-cfdf23490091 Version: $LATEST  
END RequestId: 987264ab-9484-4dcd-bc6d-cfdf23490091  
REPORT RequestId: 987264ab-9484-4dcd-bc6d-cfdf23490091 Duration: 1.96 ms Billed Duration: 2 ms Memory Size: 128 MB Max Memory Used: 16 MB

Let’s explore working with predefined AWS events. For instance, SQS can send an event when new messages are ready for consumption, and this process doesn’t require acknowledgement. The handler might look like this:

use aws\_lambda\_events::sqs::SqsEvent;  
use lambda\_runtime::{service\_fn, LambdaEvent, Error};  
  
async fn handler(event: LambdaEvent<SqsEvent>) -> Result<(), Error> {  
 Ok(println!("{event:?}"))  
}  
  
#[tokio::main]  
async fn main() -> Result<(), Error> {  
 lambda\_runtime::run(service\_fn(handler)).await  
}

While this doesn’t do anything extraordinary, it shows how to use all the predefined events available for numerous AWS integrations. These are accessible in the aws\_lambda\_events crate. Indeed, we need to install them. For this example, I have included only the SQS feature among the available options.

[package]  
name = "metrics-mingle-api"  
version = "0.1.0"  
edition = "2021"  
  
[[bin]]  
name = "metrics-mingle-api"  
path = "main.rs"  
  
[dependencies]  
aws\_lambda\_events = { version = "0.13.0", default-features = false, features = ["sqs"] }  
lambda\_runtime = { version = "0.9.0" }  
serde\_json = { version = "1.0.108" }  
tokio = { version = "1.35.1" }

To complete our demonstration, let’s build, tag, and push a new image, and then update our Lambda function using BASH commands:

# build, tag and push  
docker build -t metrics-mingle-api .  
docker tag metrics-mingle-api:latest 868404297900.dkr.ecr.eu-west-1.amazonaws.com/metrics-mingle-api:202312301612  
docker push 868404297900.dkr.ecr.eu-west-1.amazonaws.com/metrics-mingle-api:202312301612  
  
# update function code  
aws lambda update-function-code --function-name metrics-mingle-api --image-uri 868404297900.dkr.ecr.eu-west-1.amazonaws.com/metrics-mingle-api@sha256:edd2c9d4fd3066bb13395b9eeee321bd784efcf94d2bb8f3ecb0f757992e0ef2

Now, it’s time to shift our focus to a more practical application where AWS Lambda truly excels: using AWS API Gateway to convert incoming HTTP traffic into Lambda events. This functionality allows us to host a web API entirely within Lambda.

The first step is to include apigw in the supported features of the aws\_lambda\_events crate. We also need to install the http crate. With these done, we can write code to echo the body of an HTTP request:

use aws\_lambda\_events::apigw::{ApiGatewayProxyRequest, ApiGatewayProxyResponse};  
use lambda\_runtime::{service\_fn, LambdaEvent, Error};  
use http::HeaderMap;  
  
async fn handler(event: LambdaEvent<ApiGatewayProxyRequest>) -> Result<ApiGatewayProxyResponse, Error> {  
 Ok(ApiGatewayProxyResponse{  
 status\_code: 200,  
 is\_base64\_encoded: false,  
 headers: HeaderMap::default(),  
 multi\_value\_headers: HeaderMap::default(),  
 body: match event.payload.body {  
 Some(text) => Some(text.into()),  
 None => None,  
 },  
 })  
}  
  
#[tokio::main]  
async fn main() -> Result<(), Error> {  
 lambda\_runtime::run(service\_fn(handler)).await  
}

After writing the code, the next step is packaging and deploying it. For this, we need some infrastructure to connect API Gateway with AWS Lambda. Let’s add a few lines in Terraform:

resource "aws\_apigatewayv2\_api" "api" {  
 name = "metrics-mingle-api"  
 protocol\_type = "HTTP"  
 target = aws\_lambda\_function.api.arn  
}  
  
resource "aws\_lambda\_permission" "api\_gw" {  
 action = "lambda:InvokeFunction"  
 principal = "apigateway.amazonaws.com"  
 function\_name = aws\_lambda\_function.api.function\_name  
 source\_arn = "${aws\_apigatewayv2\_api.api.execution\_arn}/\*"  
}

Once everything is set up and the new Lambda function is updated, we can use curl to test our new echo HTTP server:

curl --silent \  
 --location 'https://x73275543.execute-api.eu-west-1.amazonaws.com/some/path' \  
 --header 'Content-Type: text/plain' \  
 --data 'test-me'

This time, the cold start took only 31ms, and each request required just 1ms, a significant improvement over my previous deployments using Python and Mangum.

The current method for handling HTTP load is already good, but there’s room for improvement, especially when focusing on REST API workloads. Manually matching requests for methods, endpoints, or query parameters isn’t feasible for me. An alternative is using the axum and lambda\_http crates, which let me abandon previous few crates. Here’s the final version of Cargo.toml:

[package]  
name = "metrics-mingle-api"  
version = "0.1.0"  
edition = "2021"  
  
[[bin]]  
name = "metrics-mingle-api"  
path = "main.rs"  
  
[dependencies]  
axum = { version = "0.7.3", default\_features = false, features = ["json"] }  
lambda\_http = { version = "0.9.0", default\_features = false, features = ["apigw\_http"] }  
serde\_json = { version = "1.0.108", default\_features = false }  
tokio = { version = "1.35.1", default\_features = false }

With this setup, handling any route becomes more intuitive. The code now exposes two endpoints for echoing the received body payload. The first simply outputs it, while the second wraps it around a named path parameter.

use axum::{Json, Router, routing, extract::Path};  
use lambda\_http::{self, Error};  
use serde\_json::{json, Value};  
  
async fn echo(Json(body): Json<Value>) -> Json<Value> {  
 Json(body)  
}  
  
async fn echo\_named(Path(name): Path<String>, Json(body): Json<Value>) -> Json<Value> {  
 Json(json!({ name: body}))  
}  
  
#[tokio::main]  
async fn main() -> Result<(), Error> {  
 let app = Router::new()  
 .route("/echo", routing::post(echo))  
 .route("/echo/:name", routing::post(echo\_named));  
  
 lambda\_http::run(app).await  
}

Deploying and testing the code shows that it functions as expected when we call predefined endpoints using curl. This time, specifying JSON as the content type is mandatory, as enforced by the axum crate.

curl --silent \  
 --location 'https://x73275543.execute-api.eu-west-1.amazonaws.com/echo' \  
 --header 'Content-Type: application/json' --data '{"test":"me"}'  
  
curl --silent \  
 --location 'https://x73275543.execute-api.eu-west-1.amazonaws.com/echo/it' \  
 --header 'Content-Type: application/json' --data '{"test":"me"}'

While the previous solution works, I have reservations about the feedback loop during development. What if we could work with our API in a way that allows us to try out changes immediately after making them? Let’s build it! To achieve this, we will host a local development server as a separate binary file. This server will share the same application code as the previous lambda executable, defining all routes. At the end we have two executables.

// app.rs  
use axum::{Json, Router, routing, extract::Path};  
use serde\_json::{json, Value};  
  
async fn echo(Json(body): Json<Value>) -> Json<Value> {  
 Json(body)  
}  
  
async fn echo\_named(Path(name): Path<String>, Json(body): Json<Value>) -> Json<Value> {  
 Json(json!({ name: body}))  
}  
  
pub fn build\_app() -> Router {  
 Router::new()  
 .route("/echo", routing::post(echo))  
 .route("/echo/:name", routing::post(echo\_named))  
}  
  
// lambda.rs  
mod app;  
  
use crate::app::build\_app;  
use lambda\_http::{self, Error};  
  
#[tokio::main]  
async fn main() -> Result<(), Error> {  
 lambda\_http::run(build\_app()).await  
}  
  
// server.rs  
mod app;  
  
use crate::app::build\_app;  
use tokio::net::TcpListener;  
  
#[tokio::main]  
async fn main() {  
 let listener = TcpListener::bind("0.0.0.0:3000").await.unwrap();  
 axum::serve(listener, build\_app()).await.unwrap();  
}

This modification requires redefining executables within the Cargo.toml file.

[package]  
name = "metrics-mingle-api"  
default-run = "lambda"  
version = "0.1.0"  
edition = "2021"  
  
[[bin]]  
name = "lambda"  
path = "lambda.rs"  
  
[[bin]]  
name = "server"  
path = "server.rs"  
  
[dependencies]  
axum = { version = "0.7.3", default\_features = false, features = ["json", "tokio", "http1"] }  
lambda\_http = { version = "0.9.0", default\_features = false, features = ["apigw\_http"] }  
serde\_json = { version = "1.0.108", default\_features = false }  
tokio = { version = "1.35.1", default\_features = false }

With these changes, we can now run a local server using cargo watch, continue running cargo lambda, or fully package the deployment with docker build.

# run local HTTP Server on localhost:3000  
cargo watch -- cargo run --bin server  
  
# run local Lambda Scheduler on localhost:9000  
cargo lambda watch -a 127.0.0.1  
  
# package deployment  
docker build -t metrics-mingle-api .

In this article we learnt how to write and deploy an AWS Lambda function written in Rust. I have a feeling it is a game changer in the way how highly performant code can run in the Cloud where you are charge for each bit or millisecond. Are you ready to give it a try?