A Little Rust-y: Using Rust Interop To Boost Performance of Existing Apps (In Any Language)

Introduction

This document provides an outline to walk through our "A Little Rust-y" demo. In this demo we will start with an existing node.js application, intended to deploy to an AWS lambda environment. We will explore the functionality and performance of the application, use profiling tools to identify bottlenecks, and then surgically replace a small part of the application with some Rust code, called via the node.js program via interop. Afterward we'll revisit the performance and celebrate the improvements!

This demo does not assume that you have any familiarity with or experience with Rust; we'll cover the basics that you need to know. By the end of this project you should have enough familiarity to understand how you might apply these techniques to your own application, regardless of what language it's written in. And you will see that it's possible to achieve major performance wins (and cost reductions!) with a tiny bit of Rust, without needing to re-write your whole application!

The source code for this demo can be found at
https://github.com/momentohq/demo-nodejs-rust-lambda . The material is broken into milestones so that we can checkpoint our progress along the way. The `main` branch contains the example node.js application that we will use as a starting point - Milestone 1. As we walk through the demo we will write the code for the subsequent milestones collaboratively so that you can get a hands-on experience. However, a finished, working example of each milestone is available in the git repo in branches named `milestone2`, `milestone3`, etc. If you have trouble with any of the sections and just want to jump to one of those checkpoints, feel free to `git checkout` the appropriate branch.

Overview of the Existing Application

The application we will be working on is intended to mimic a real-world bulk ingestion job. The idea is that you might have a production application that consumes data from an external system, and you periodically get bulk updates of that data that you need to ingest into a database or cache for consumption by other parts of your system.

In this demo we will consume some sample weather data and ingest it into a distributed cache. The data is available via HTTP in a large JSONL file, where each line contains the latest weather data for a given city. The file contains 200,000 data points from cities all around the world. We will use an AWS Lambda to retrieve the data file, parse some relevant data for each city, and store it in the cache. In a real production application, presumably you would have other components that were consuming this data from the cache, but we'll be focusing on just this ingestion piece.

For this demo we will use a Momento cache for the distributed cache, but the techniques we cover here should work equally well with many other data stores that you might be ingesting your data into.

Milestone 1: Set up environment and tour the existing node.js app

To follow along with the hands-on parts of this demo, you will need the following (if you are attending a workshop led by Momento engineers, we will provide you with all of these things up front!)
- A Momento cache with sufficient throttling limits to run this code (40,000 operations per second and 2MB/s throughput limit. Please reach out to us on discord or via e-mail if you need help with these limits!)
- A Momento API key for reading and writing the cache
- A Momento Flush API key for flushing the cache

First, clone the git repo:

git clone https://github.com/momentohq/demo-nodejs-rust-lambda.git

Set up your .env file

Once you have a local copy of the code, the first thing you'll need to do is configure the `.env` file. Your hosts may provide you with one that you can drop into place; otherwise, there is an example `.env` file in the repo named `.env.EXAMPLE`. You can copy this to `.env` and edit the variables accordingly:

```
LOG_LEVEL=debug
MOMENTO_CACHE_NAME=<your-cache-name>
MOMENTO_API_KEY=<your-api-key>
MOMENTO_FLUSH_API_KEY=<your-flush-api-key>
```
Once you have configured the `.env` file, we can run the sample program to get familiar with what the code does.

If you'd like to take a peek at the weather data file to see what the input looks like, you can find it in the repo in the `data` directory, or on github here:

https://github.com/momentohq/demo-nodejs-rust-lambda/blob/a8253ea784c214f4f766fbab8116 8515b7300d44/data/weather_16.json.gz

The code we're interested in to kick things off is all contained in the `lambdas/nodejs` directory:

```
cd lambdas/nodejs
```

Run the `sample-cache-data.ts` script to see that the cache is empty

The package.json file contains script targets for all of these programs, so you can run them easily via npm run:

```
npm run sample-cache-data
```

You should see some output that looks like this; notice that the program was not able to retrieved cached weather data for any of the cities:

```
> demo-pure-nodejs-lambda@1.0.0 sample-cache-data
> tsc && node dist/sample-cache-data.js

2024-10-10T17:19:16.233Z info: Instantiating Momento CacheClient
2024-10-10T17:19:16.393Z info: San Francisco weather: undefined
2024-10-10T17:19:16.437Z info: Seattle weather: undefined
2024-10-10T17:19:16.487Z info: Portland weather: undefined
2024-10-10T17:19:16.540Z info: Austin weather: undefined
2024-10-10T17:19:16.546Z info: PureNodeJsMomentoCache closed
```

Run the `local-ingest.ts` script to perform the ingestion and populate the cache

Now we'll run the local-ingest script to execute the ingestion. For this one it can be useful to use the time command so that we can observe how long it takes to execute.

We will be ingesting 200,000 items into the cache, so network latency is a big factor. The execution time may vary quite a bit depending on your network connection.

```
time npm run local-ingest
```

You should see some output like this:

```
> demo-pure-nodejs-lambda@1.0.0 local-ingest
> tsc && node dist/local-ingest.js

2024-10-10T17:22:25.981Z info: Instantiating Momento CacheClient
2024-10-10T17:22:26.003Z info: Initializing weather data reader
2024-10-10T17:22:26.679Z info: GzippedUrlLineReader: read all lines
2024-10-10T17:22:26.679Z info: Weather data reader initialized
2024-10-10T17:22:26.679Z info: Starting 1000 workers
2024-10-10T17:22:26.679Z info: Waiting for all workers to complete
2024-10-10T17:22:46.425Z info: Worker 996 finished
…
2024-10-10T17:22:46.483Z info: Worker 467 finished
2024-10-10T17:22:46.483Z info: All workers completed
2024-10-10T17:22:46.483Z info: Weather data reader closed
2024-10-10T17:22:46.484Z info: Cached 209579 weather data points
npm run local-ingest  26.90s user 2.43s system 135% cpu 21.721 total
```

You can configure the `LOG_LEVEL` to either `info` or `debug` via the `.env` file; if you are logging at `debug` you will see quite a bit more information about the individual data points that were saved to the cache.

If you have a reasonably fast residential network connection (fiber), this program may complete in about 20 seconds. For slower network connections it will take longer.

Run the sample-cache-data.ts script again to see that the cache is populated

Let's run the sample-cache-data program again, to verify that the cache was populated:

```
npm run sample-cache-data
```

Notice that now we do see the data points for the cities! (The temperatures are in Kelvin, because science!)

```
> demo-pure-nodejs-lambda@1.0.0 sample-cache-data
> tsc && node dist/sample-cache-data.js

2024-10-10T17:37:31.161Z info: Instantiating Momento CacheClient
2024-10-10T17:37:31.240Z info: San Francisco weather:
{"minTemp":297.279,"maxTemp":297.279}
```

```
2024-10-10T17:37:31.281Z info: Seattle weather:
{"minTemp":282.04,"maxTemp":284.26}
2024-10-10T17:37:31.322Z info: Portland weather:
{"minTemp":285.37,"maxTemp":287.59}
2024-10-10T17:37:31.364Z info: Austin weather:
{"minTemp":282.04,"maxTemp":287.15}
2024-10-10T17:37:31.366Z info: PureNodeJsMomentoCache closed
```

Run the flush-cache.ts script to flush the cache

Now, just to help with future development and debugging, let's see how to flush the cache and verify that the data was cleared:

```
npm run flush-cache

> demo-pure-nodejs-lambda@1.0.0 flush-cache
> tsc && node dist/flush-cache.js

2024-10-10T17:39:30.787Z info: Instantiating Momento CacheClient
2024-10-10T17:39:31.212Z info: Cache flushed
2024-10-10T17:39:31.215Z info: PureNodeJsMomentoCache closed

npm run sample-cache-data

> demo-pure-nodejs-lambda@1.0.0 sample-cache-data
> tsc && node dist/sample-cache-data.js

2024-10-10T17:39:34.228Z info: Instantiating Momento CacheClient
2024-10-10T17:39:34.308Z info: San Francisco weather: undefined
2024-10-10T17:39:34.351Z info: Seattle weather: undefined
2024-10-10T17:39:34.393Z info: Portland weather: undefined
2024-10-10T17:39:34.439Z info: Austin weather: undefined
2024-10-10T17:39:34.441Z info: PureNodeJsMomentoCache closed
```

(Optional) Deploy the app to AWS Lambda and run it there

This code is intended to simulate a recurring data ingestion job that might be part of a production application stack. You can imagine the ingestion being triggered on a schedule, or via an event notification that fires when new data is available. AWS Lambda can be a great choice for hosting those kinds of relatively short-lived jobs, so that's how this demo is structured.

The `infrastructure` directory at the root of the repo contains an AWS CDK app that can be used to deploy the app to AWS Lambda.

To deploy to Lambda you will need an AWS account, and you will need to have your AWS credentials configured so that you can successfully run CLI commands such as `aws s3 ls`. We won't be running the Lambda more than a few times, and it doesn't run for very long, so hopefully this demo will fit within the AWS free tier. However, if you don't have an AWS account or you don't want to risk incurring any costs at all, feel free to just follow along as the presenters demonstrate this step.

A note on hardware architecture

The Rust code that we will be writing soon needs to be targeted at a specific hardware architecture (ARM64 vs x86_64). Since we are running this app both locally and on AWS Lambda, the easiest path is to deploy to lambda using the architecture that most closely matches your local dev machine. E.g., if you're on a Mac with Apple silicon, you should probably go with ARM64. If you're on an Intel/AMD machine then choose x86_64.

You can configure which architectured will be used for the Lambda here:

https://github.com/momentohq/demo-nodejs-rust-lambda/blob/8e05b9955b5a220cafab1ee41b8aaa3e77c7ca26/infrastructure/bin/infrastructure.ts#L52

cdk deploy

To deploy the Lambda (again, assuming you have already configured your AWS credentials), run the following command (from inside the `infrastructure` directory):

`npm run deploy`

You may be prompted to accept the changes that CDK detects it will be deploying. After a successful deploy you will see something like this:

```
DemoNodejsRustLambdaStack: deploying... [1/1]

✅   DemoNodejsRustLambdaStack (no changes)

✨   Deployment time: 0.22s
```

Running the Lambda via the AWS console

After a successful CDK deploy, you can visit the Lambda service page in the AWS console, and navigate to the `DemoPureNodeJsLambda` function. Then click on the `Test` tab, and then the `Test` button, to invoke the Lambda:

The lambda should take around 80-90 seconds to execute. When it completes you can see the run duration here:

```
Executing function: succeeded (logs ↗)
▼ Details

The area below shows the last 4 KB of the execution log.

{
  "statusCode": 200,
  "headers": {
    "Content-Type": "application/json",
    "Access-Control-Allow-Origin": "*"
  },
  "body": "{}"
}
```

Summary

Code SHA-256	Execution time
O/kcJPBJbTYv1Pk8qfjZvEo8YixD4UnLcuBFtXAuBTk=	2 minutes ago
Request ID	Init duration
ca174552-fe9e-460f-aa6f-f3dbb462f48f	363.65 ms
Duration	Billed duration
84012.89 ms	84013 ms
Resources configured	Max memory used
8192 MB	1439 MB

And we can go back to the `lambdas/nodejs` dir and run `npm run sample-cache-data` again to validate that the data was ingested into the cache successfully.

90 seconds isn't necessarily terrible for ingesting 200,000 items into the cache… but we have also configured a pretty large instance size (8GB) for this Lambda. We can probably do better; let's see where the code is spending its time!

Use profiling tools to identify performance hotspots in the nodejs code

Back on your laptop/dev machine; let's run the ingestion locally again, but this time with a profiler attached so that we can see where time is being spent.

There is a very simple-to-use profiling tool for node.js called `0x`. It can be used to generate a nice flame graph of the hot spots in your code. You don't even have to install anything, we can just run it with `npx`! From the `lambdas/nodejs` directory:

```
npm run build
npx 0x ./dist/local-ingest.js
```

If the ingest program takes a long time to run in your dev environment, you don't have to let this command complete. You can CTRL-C it after 30 seconds or so; when it exits you will see something like this:

```
🔥   Flamegraph generated in
file:///Users/cprice/demo-nodejs-rust-lambda/lambdas/nodejs/56610.0x/
flamegraph.html
```

Once we have the flame graph, we can open it in a browser. It should look something like this:
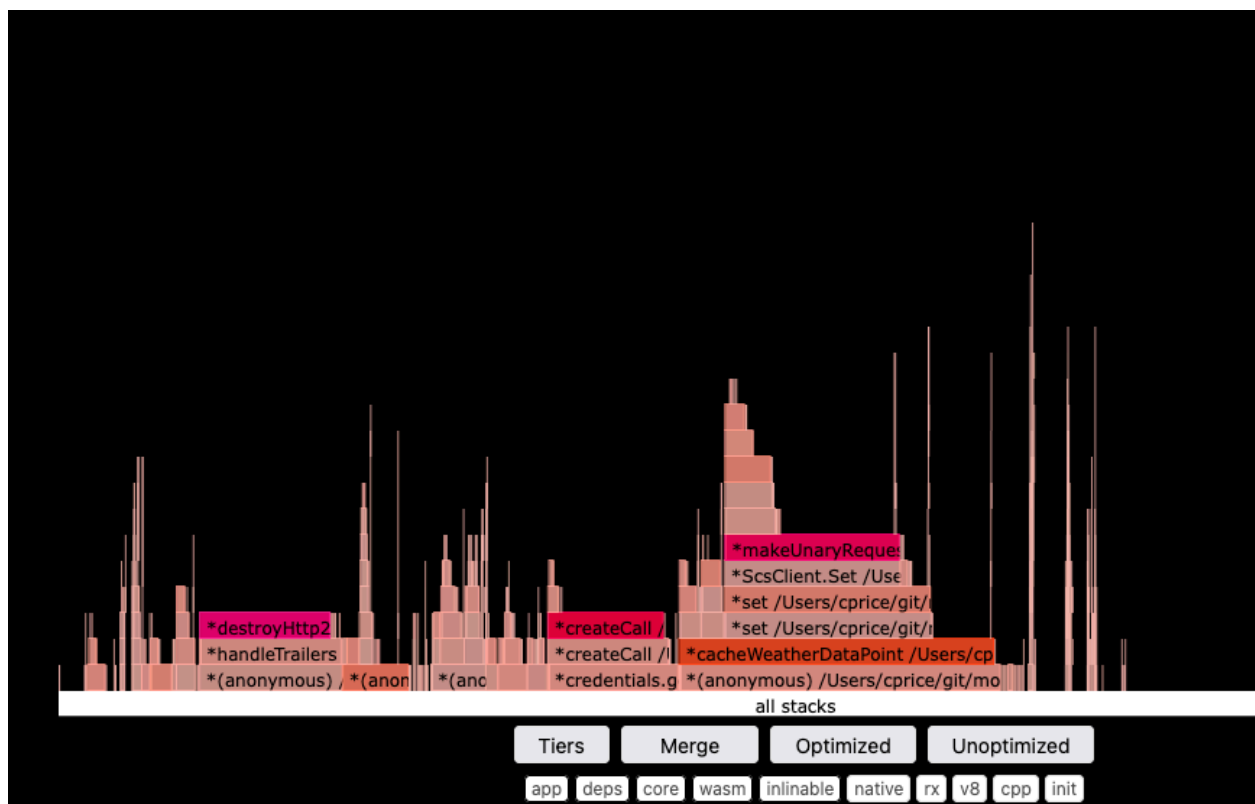
You can find some information about how to read this graph here:

https://github.com/davidmarkclements/0x/blob/master/docs/ui.md

But the tl;dr is that we are looking for the **widest** band of call stack, and that is where the app is spending the most time. In this case, if we click in a few levels, we can see that the widest band call stack by far is in the "*set" method, inside of `cacheWeatherDataPoint`. These are the network calls for the writes to the distributed cache, coming from the Momento SDK, and then going down into the `grpc-js` library:

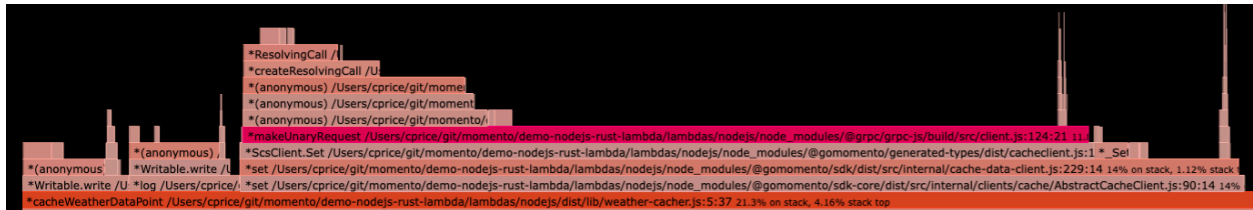Thankfully, this is a great example of a type of call stack that is likely to optimize well in Rust!

Why Rust may provide performance improvements for network calls

Here are some reasons why porting our network calls to Rust may provide perf improvements:

1. If the original application is in a language like node.js or python, there is a bottleneck because of the fact that their runtimes are effectively single-threaded for executing user code. Their event loops can become bottlenecked and max out a single CPU. Rust supports true parallelism across CPU cores, and thus **Rust can better take advantage of available CPU**.
2. Rust networking libraries tend to be written with a huge emphasis on performance. The engineers behind these libraries have often put a lot of effort into optimizing, and especially into eliminating unnecessary memory allocations.
3. Rust compiles to native code, so it is closer to the hardware. Languages like node.js and python are interpreted, and they have an execution runtime that sits between the user code and the hardware.

With that, let's move on to the Rust sections of this workshop!

Milestone 2: Rust Hello World

The completed code for this milestone can be found in the [milestone2 branch](#) of the repo.

In this milestone we will focus on just building out a simple "Hello World" Rust project and learn a few basic concepts. Once we have that working we can start taking steps towards the caching tasks.

We'll start with building out the scaffolding for a Rust project.

Creating a cargo workspace

`cargo` is the build tool that you will use to build and run your Rust programs. Each individual Rust project (library, CLI app, etc.) is called a "crate". We will ultimately be creating 2 crates; one to contain our Momento implementation of the distributed cache facade, and a second one that uses a framework called "napi-rs" to make it possible to call our distributed cache Rust functions from node.js.

We will start off by creating a `cargo` "workspace", which is a way to group multiple crates together in the same directory structure. (For more info on `cargo` workspaces, see https://doc.rust-lang.org/book/ch14-03-cargo-workspaces.html)

To create our cargo workspace, we just need to make a directory and then create a file inside of it called `Cargo.toml`:

```
mkdir lambdas/rust-workspace
cd lambdas/rust-workspace
cat <<EOT >> Cargo.toml
[workspace]
resolver = "2"

members = [
  "momento_cache_facade",
]
EOT
```

This just tells cargo that this directory is a workspace, and that it will have a subdir called `momento_cache_facade` that will contain our first crate.

Adding a library crate to our workspace

We can use cargo new to create our first library crate:

```
cargo new momento_cache_facade --lib
```

This will create the `momento_cache_facade` dir, with its own `Cargo.toml` file and some sample library code.

We can now run `cargo build` from either the workspace dir or the `momento_cache_facade` dir, to build the library.

Adding a bin program to the library crate

Next we will add a CLI app to our library crate, so that we can write some toy code to test various things and consume our library code. To do this, we just need to add a `bin` section to the crate's `Cargo.toml` file:

```
[[bin]]
name = "hello_momento_cache"
path = "src/bin/hello.rs"
```

Then we need to create the `src/bin/hello.rs` file, and define a `main` function in it:

```
fn main() {
    println!("Hello, world!");
}
```

Now we should be able to do `cargo run` to run the hello app.

Adding a struct to the library

Structs in Rust are somewhat similar to classes in OO languages. We'll start off by defining a `MomentoCacheFacade` struct and giving it a few arbitrary fields. Replace the contents of the `src/lib.rs` file with:

```
pub struct MomentoCacheFacade {
    pub foo: String,
    pub bar: u32
}
```

Now, from our hello.rs app, we can instantiate an instance of this struct:

```
use momento_cache_facade::MomentoCacheFacade;

let foo = "Hello!"
let cache_facade = MomentoCacheFacade{
    foo: foo.to_string(),
    bar: 42
};
```

Next we'd like to be able to print it, for debugging purposes. In Rust, the `println!` macro supports printing a debug representation of a struct using the `{:?}` format string, like this:

```
println!("The cache facade instance is: {:?}", cache_facade);
```

However, any object that you wish to be able to use with the `{:?}` format string must implement Rust's `Debug` trait. A trait in Rust is like an interface in many OO languages. The `Debug` trait just specifies that the struct must implement a special `fmt` function that will produce a string representation of the struct for debugging.

Rust can automatically generate the implementation of the `Debug` trait (and other common traits!) for simple structs like ours. To do that, we just need to decorate the `struct` declaration with the `derive` macro:

```
#[derive(Debug)]
pub struct MomentoCacheFacade {
…
```

Now we can print out our instance of the struct in our `hello.rs` program.

Side note: when we run `cargo build` or `cargo run`, by default, the binary that is produced includes debug symbols. For either of these commands you can add the `-release` flag, which will optimize the binary for production environments. This can make a very big difference in performance so make sure that all of your production builds specify the `-release` flag!

Let's make one last set of changes to our struct for the sake of learning a little more Rust. We mentioned earlier that structs are kind of like classes; to implement methods for the struct we use the impl expression. This also allows us to create a factory function so that we don't need to make the fields of the struct public:

```
#[derive(Debug)]
pub struct MomentoCacheFacade {
     foo: String,
     bar: u32
}

impl MomentoCacheFacade {
     pub fn new(foo: String, bar: u32) -> MomentoCacheFacade {
         MomentoCacheFacade {
              foo,
              bar
         }
     }

     pub fn get_foo(&self) -> String {
         self.foo.clone()
     }
}
```

Any `fn` inside the impl that does not have a `self` argument is a static/class function. Functions that have a `self` argument are like instance methods.

This is a good place to wrap up our initial Rust "Hello World" explorations. Let's move on to interacting with the Momento cache!

Milestone 3: Pure Rust Cache Facade

The completed code for this milestone can be found in the [milestone3 branch](#) of the repo.

Now let's modify our struct so that it actually provides some functions for interacting with a Momento cache.

Adding Rust dependencies

First let's add a few dependencies. We'll need the `momento` crate in order to communicate with Momento. Let's also add a couple of crates for logging, so that we can write log statements at both `info` and `debug` levels.

To add dependencies we can use `cargo add`:

```
cd lambdas/rust-workspace/momento_cache_facade
cargo add momento
cargo add log
cargo add colog
```

This is just a helpful tool for updating our Cargo.toml file. If you look at that file now, you will see these new dependencies added in the `[dependencies]` section.

Configure logging

Now that we've added these dependencies, let's modify lib.rs to add a function for initializing logging. This is a good chance to get some hands-on experience with Rust's incredibly useful `match` expression (which happens to also be the best way to avoid `expect/unwrap`):

```
pub fn initialize_logging(log_level: &str) {
    let mut default_builder = colog::default_builder();
    let log_builder = match log_level {
```

```
        "debug" =>
    default_builder.filter_level(log::LevelFilter::Debug),
        "info" =>
    default_builder.filter_level(log::LevelFilter::Info),
        _ => default_builder.filter_level(log::LevelFilter::Info),
    };
    // this logger is initialized globally for the Rust process. In
an
    // AWS Lambda environment, the initialized state can persist
    // across lambda invocations, so we need to handle the case
where
    // it has already been initialized
    match log_builder.try_init() {
        Ok(_) => info!("Logging initialized"),
        Err(e) => warn!("Failed to initialize logging; this may
    just indicate that logging is already initialized (e.g. in a
    warm Lambda container): {}", e),
    }
}
```

Construct Momento CacheClient

Then let's modify the `MomentoCacheFacade` struct to construct a Momento client, and provide some functions for storing and retrieving cache items:

```
#[derive(Debug)]
pub struct MomentoCacheFacade {
    cache_name: String,
    cache_client: CacheClient,
}

impl MomentoCacheFacade {
    pub fn new(cache_name: String, momento_api_key: String) ->
MomentoCacheFacade {
        MomentoCacheFacade {
            cache_name,
            cache_client: CacheClient::builder()
                .default_ttl(Duration::from_secs(60 * 10))

    .configuration(configurations::InRegion::latest())
                .credential_provider(

    CredentialProvider::from_string(momento_api_key)
```

```
                            .expect("Failed to get API key from
    environment variable"))
                    .with_num_connections(10)
                    .build()
                    .expect("Failed to build cache client"),
        }
    }

    pub async fn store(&self, key: &str, value: &str) {
        debug!("Rust MomentoCacheFacade storing key {} with value
    {}", key, value);
        self.cache_client.set(&self.cache_name, key,
    value).await.expect("Failed to store value in cache");
    }

    pub async fn retrieve(&self, key: &str) -> Option<String> {
        debug!("Rust MomentoCacheFacade retrieving key {}", key);
        let response = self.cache_client.get(&self.cache_name,
    key)
            .await
            .expect("Failed to retrieve value from cache");

        match response {
            GetResponse::Hit { value } =>
    Some(value.try_into().expect("Failed to convert value to
    string")),
            GetResponse::Miss => None
        }
    }
}
```

Some things to note in this code:
- We're using an `Option` here as the return type for the `retrieve` fn.
- We're using the `info!` and `debug!` macros from the log crate for logging.

Update hello.rs CLI app to exercise new struct API

Now let's update our hello.rs CLI app to consume the new version of this struct. We'll need two more dependencies to wrap this up:
- `dotenv`, just to make it easy to access the same configuration / environment variables from the .env file that we've been using so far.

- `tokio`, the most popular asynchronous runtime for Rust. Rust supports `async`/`await` keywords as part of the core language, but you have to add a dependency on an `async` runtime such as `tokio` in order to actually write and call `async` functions.

When adding a dependency on a Rust crate, sometimes the crates publish multiple `features` that you can opt into or opt out of. In the case of `tokio`, there are a few non-default features that we need. The easiest way to make sure we get everything we need is to request the `full` feature:

```
cargo add dotenv
cargo add tokio --features full
```

Now we have all the deps we need to update our `hello.rs` to actually interact with the Momento cache via our struct:

```
use log::info;
use momento_cache_facade::{initialize_logging, MomentoCacheFacade};

#[tokio::main]
async fn main() {
    dotenv::from_path("../../../.env").expect("Failed to load .env
file");
    let momento_cache_name =
dotenv::var("MOMENTO_CACHE_NAME").expect("Failed to get
MOMENTO_CACHE_NAME from .env file");
    let momento_api_key =
dotenv::var("MOMENTO_API_KEY").expect("Failed to get MOMENTO_API_KEY
from .env file");
    let log_level = dotenv::var("LOG_LEVEL").expect("Failed to get
LOG_LEVEL from .env file");
    initialize_logging(&log_level);
    let cache_facade = MomentoCacheFacade::new(momento_cache_name,
momento_api_key);
    info!("Storing value in cache");
    cache_facade.store("foo", "FOO").await;
    let value = cache_facade.retrieve("foo").await;
    info!("Retrieved value from cache: {:?}", value);
}
```

We can test this using `cargo run`.

Milestone 4: Adding the napi-rs Rust module

The completed code for this milestone can be found in the [milestone4 branch](#) of the repo.

Okay, so now we have a working Rust crate that has the basic APIs we need for implementing our `DistributedCache` interface in our node.js app. The last piece of the puzzle is to make it possible to call the Rust code from the node.js code. To do this, we use a library called `napi-rs`:

https://napi.rs/

Create second crate for napi-rs project

We will create a second crate in our rust workspace to define the `napi-rs` bindings.

We can use the `napi-rs` cli to create a skeleton project for us:

```
npx @napi-rs/cli new
```

This will start an interactive session which will prompt you for a few inputs.

- First, it will prompt you for a package name. Enter `napi_rs_momento_cache`.
- Then it will prompt you for a directory name; press enter to accept the same name.
- After this it will prompt you to select the architectures you want to build for. For this demo, you probably only need 2: one for your laptop/dev environment, and one for the AWS Lambda environment. In my case I chose `aarch64-apple-darwin` and `aarch64-unknown-linux-gnu`. If you are developing on windows, pick the appropriate windows target instead of apple; if you are on `x86_64` you can pick that instead of `aarch64`. You will need to include one linux target for deploying to AWS Lambda.
- When it prompts you to add github actions, choose "no" for now; we don't need these for our demo purposes.

After you've completed the interactive session you should have a new `napi_rs_momento_cache` directory with the skeleton napi-rs project. We need to add this to our workspace `Cargo.toml` so that it will be included in our workspace builds, and so that we can consume the `momento_cache_facade` crate from this new crate.

To do this, you'll just need to edit the `lambdas/rust-workspace/Cargo.toml` file, and add `napi_rs_momento_cache` to the `members` array.

Now, from the `lambdas/rust-workspace` directory, we should be able to do a `cargo build` and observe that it builds both of the member crates.

Consume momento_cache_facade from napi_rs crate

Next, we can edit the code in the generated `napi_rs_momento_cache` crate to make it actually consume the `momento_cache_facade`. The first thing we need to do is edit `lambdas/rust-workspace/napi_rs_momento_cache/Cargo.toml` to add a dependency on the `momento_cache_facade` crate. We will add a line to the `[dependencies]` section that looks like this:

```
momento_cache_facade = { path = "../momento_cache_facade" }
```

Then, we can modify the napi_rs_momento_cache/src/lib.rs file to look like this:

```
#![deny(clippy::all)]

#[macro_use]
extern crate napi_derive;

#[napi]
pub struct NapiRsMomentoCache {
  momento_cache: momento_cache_facade::MomentoCacheFacade
}

#[napi]
impl NapiRsMomentoCache {
  #[napi(factory)]
  pub fn create(momento_api_key: String, cache_name: String) -> Self
{
      NapiRsMomentoCache {
          momento_cache:
momento_cache_facade::MomentoCacheFacade::new(cache_name,
momento_api_key)
      }
  }

  #[napi]
  pub async fn store(&self, key: String, value: String) {
      self.momento_cache.store(&key, &value).await
  }

  #[napi]
  pub async fn retrieve(&self, key: String) -> Option<String> {
      self.momento_cache.retrieve(&key).await
```

```
    }
}
```

As you can see, this is a very thin wrapper around the `MomentoCacheFacade` struct from our original crate. The only purpose of this code is to allow napi-rs to generate the final binary output that allows us to call this code from node.js.

To build this, we can go into the `lambdas/rust-workspace/napi_rs_momento_cache` directory and run `npm run build`.

If you do this now, you will get an error from cargo that says "`error[E0425]: cannot find function `execute_tokio_future` in module `napi::bindgen_prelude`"`. This is because we defined `async` functions in our napi-rs API, and once again, async things in Rust require an async runtime like `tokio`. In this case we need to update our napi dependency (in `lambdas/rust-workspace/napi_rs_momento_cache/Cargo.toml`) to add the `tokio_rt` feature to the dependency. After doing this the dependency line will look like this:

```
napi = { version = "2.12.2", default-features = false, features = ["napi4", "tokio_rt"] }
```

For more information about async support in napi-rs, see their docs; e.g.:

https://napi.rs/docs/concepts/async-fn

Now we should be able to run `npm run build` successfully. When it completes, you should see some generated `index.d.ts` and `index.js` files, which contain the node.js-compatible type definitions and javascript bindings, as well as a binary file like `napi_rs_momento_cache.darwin-arm64.node` that contains the actual compiled Rust code.

One more thing to notice when you run `npm run build`: if you look at the console output you will see that it is passing `-release` when it calls cargo to compile the Rust code. This is very important for performance! But thankfully, `napi-rs` does the right thing by default.

Now we're ready to try calling this from node.js!

Milestone 5: Running the ingest locally with the napi-rs version

The completed code for this milestone can be found in the [milestone5 branch](#) of the repo.

To test this out locally, we'll head back to the nodejs project and create a new implementation of our `DistributedCache` interface that uses our napi-rs crate.

Add a dependency on our napi_rs project

The first thing we need to do is add an npm dependency on our napi-rs project, which we can do like this:

```
cd lambdas/nodejs
npm install ../rust-workspace/napi_rs_momento_cache
```

That will result in the package.json file being updated to include:

```
"napi_rs_momento_cache":
"file:../rust-workspace/napi_rs_momento_cache",
```

Create Rust version of DistributedCache interface

Now we should be able to reference the Rust struct that we annotated with `#[napi]` as though it were a javascript class. We'll create a file in `lambdas/nodejs/lib/distributed/cache` called `rust-momento-cache.ts`, and its contents will look like this:

```
import {DistributedCache} from './distributed-cache';
import {NapiRsMomentoCache} from 'napi_rs_momento_cache';

export class RustMomentoCache implements DistributedCache {
  private readonly cache: NapiRsMomentoCache;

  constructor(logLevel: string, momentoApiKey: string, cacheName:
string) {
      this.cache = NapiRsMomentoCache.create(logLevel, momentoApiKey,
cacheName);
  }

  store(key: string, value: string): Promise<void> {
      return this.cache.store(key, value);
  }

  async retrieve(key: string): Promise<string | undefined> {
      const result = await this.cache.retrieve(key);
      return result === null ? undefined : result;
  }

  flush(): Promise<void> {
```

```
    throw new Error('Flush not yet implemented for
NapiMomentoCache');
  }

  async close(): Promise<void> {
    // nothing to do
  }
}
```

Create a script to run locally

Then we'll make a copy of `local-ingest.ts` called `local-ingest-rust.ts`, and instantiate this new `RustMomentoCache` instead of the `PureNodejs` one:

```
const cache = new RustMomentoCache(envVars.LOG_LEVEL,
envVars.MOMENTO_API_KEY, envVars.MOMENTO_CACHE_NAME);
await cacheWeatherData(logger, cache);
```

And add an entry to the `scripts` section of `package.json` to allow us to easily run it:

```
"napi_rs_momento_cache":
"file:../rust-workspace/napi_rs_momento_cache",
```

Now we are ready to run it, via `time npm run local-ingest-rust`.

Depending on your network latency, you may observe that this version runs as much as 2x faster than the pure nodejs version. Big win, with very little code!
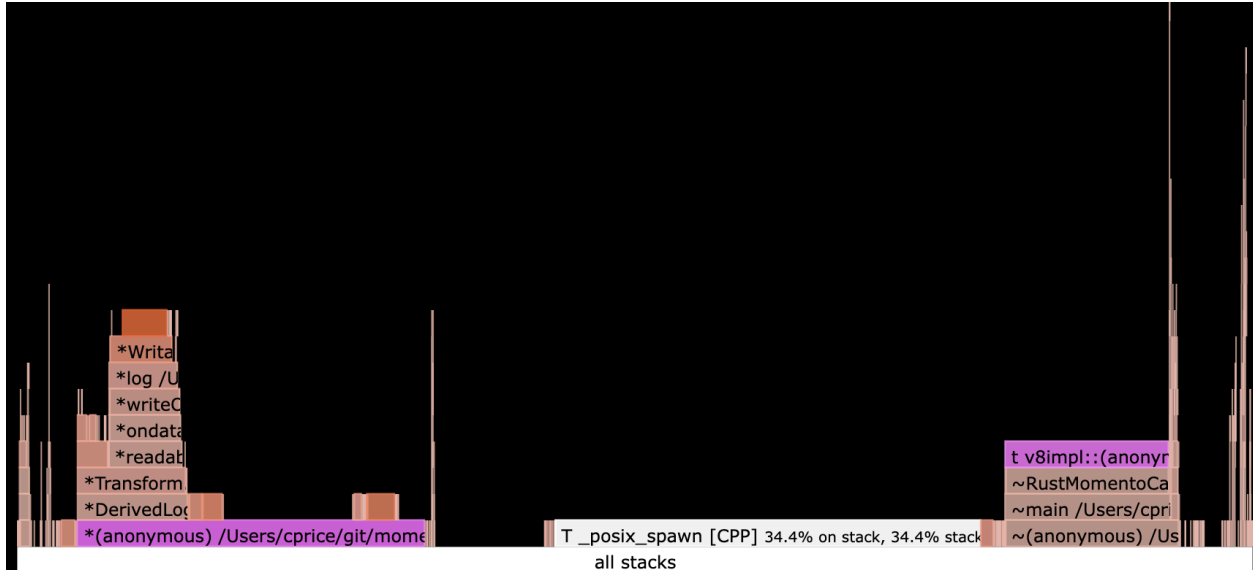
Revisit our flame graph

We can also revisit our flame graph at this point to see how the characteristics may have changed:

```
npm run build
npx 0x ./dist/local-ingest-rustnp.js
```

Sifting through this updated flame graph the main observation is that we have eliminated most of the obvious JS stack frames. We can see now that the widest stacks show up as "CPP", meaning that they are calling through napi to the native binary code. There is one JS stack left that we could theoretically poke at, which has to do with logging. But it appears that we've successfully moved the most dominant stack frames that we saw in our previous flame graph across to the native code.

Okay, looks good locally. But now for the real test; how does it look when we run it on AWS Lambda?

Milestone 6: (Optional) Deploying the napi-rs version to AWS Lambda

The completed code for this milestone can be found in the [milestone6 branch](#) of the repo.

When we deploy to AWS lambda, one thing that we know will be significantly different is that the network latencies will be much lower. The minimum latency from a Lambda deployed in us-west-2 to the Momento cache in us-west-2 will be almost negligible compared to the latency of running from a laptop / dev environment. Therefore, any performance improvements we have realized that make the app more efficient with memory and CPU may actually be more pronounced in AWS Lambda than on the laptop.

So let's modify our build environment and CDK code to deploy a second Lambda, using the napi-rs integration.

Build the Linux binary

An important first step is that we need to build the rust binary (.node file) using the Lambda target architecture (e.g. aarch64-unknown-linux-gnu, or the x86_64 equivalent if you're targeting x86 architecture).
We can pass an extra `--target` flag to the napi build command in order to specify the architecture. To make this easier/more repeatable we can add a `script` to the `lambdas/rust-workspace/napi_rs_momento_cache /package.json` file, e.g.:

```
    "build": "napi build --platform --release",
    "build-linux": "npx napi build --platform --release --target
aarch64-unknown-linux-gnu",
```

After that, we can run the napi build (from the `lambdas/rust-workspace/napi_rs_momento_cache directory`):

```
npm run build-linux
```

After this, you should see a new .node file in the directory (e.g. `napi_rs_momento_cache.linux-arm64-gnu.node`).

Create the Node.js Lambda handler function

The next step is to create a node.js handler function for our second lambda. To do this we'll just copy `lambdas/nodejs/pure-nodejs-handler.ts` to `lambdas/nodejs/nodejs-rust-handler.ts,` and modify the line where we instantiate the `DistributedCache` so that we use the new Rust version instead of the PureNodejs version:

```
const cache = new RustMomentoCache(envVars.LOG_LEVEL,
envVars.MOMENTO_API_KEY, envVars.MOMENTO_CACHE_NAME);
```


Add the new Lambda to the CDK code

Now we are ready to update the CDK code to add the new Lambda, and we'll just need to do a little bit of extra work when configuring it to make sure that it includes our napi module. Let's start by, in infrastructure/lib/demo-nodejs-rust-lambda-stack.ts, just making a copy of the code that we use to provision the PureNodejs lambda, and changing the names/paths to point to the new handler:

```
new cdk.aws_lambda_nodejs.NodejsFunction(this,
'DemoNodejsRustLambda', {
    functionName: 'DemoNodejsRustLambda',
    runtime: cdk.aws_lambda.Runtime.NODEJS_20_X,
    architecture: demoProps.architecture,
    entry: path.join(__dirname,
'../../lambdas/nodejs/nodejs-rust-handler.ts'),
    projectRoot: path.join(__dirname, '../../lambdas/nodejs'),
    depsLockFilePath: path.join(__dirname,
'../../lambdas/nodejs/package-lock.json'),
    handler: 'handler',
    timeout: cdk.Duration.seconds(300),
    memorySize: 8192,
    environment: {
        MOMENTO_API_KEY: demoProps.momentoApiKey,
        MOMENTO_FLUSH_API_KEY: demoProps.momentoFlushApiKey,
        MOMENTO_CACHE_NAME: demoProps.momentoCacheName,
        LOG_LEVEL: demoProps.logLevel,
    },
    bundling: {
        forceDockerBundling: true,
    },
});
```

If we try to deploy this right now, it will fail when it is trying to bundle the
`napi_rs_momento_cache` dependency, because this dependency comes from a local
directory, and the build does not know how to handle the local directory symlinks or the binary
file. Thankfully the CDK APIs give us hooks we can use to amend the build steps, so that's what
we need to do here.

Handle the packaging of `napi_rs_momento_cache` into the Lambda image

The first thing we need to do is to tell the build to basically ignore the
`napi_rs_momento_cache` during the bundling process, because we are going to package it
into the final build ourselves. To do that we just add the externalModules setting and add our
package to it:

```
bundling: {
    forceDockerBundling: true,
    externalModules: ['napi_rs_momento_cache'],
},
```

At this point you could run `npm run deploy` and the CDK deploy would complete successfully. However, if you try to run the Lambda, it will fail at runtime when it tries to load `napi_rs_momento_cache`, since we haven't added it to the package yet.

So, we'll need a few more steps to add our napi module to the packaging. Since this build is happening inside a docker container, is to mount the `napi_rs_momento_cache` directory as a volume so that it is accessible to the docker build steps inside of the container. We can do that by just adding this snippet inside of the `bundling` section of the lambda CDK code:

```
  volumes: [
          {
          hostPath: path.join(__dirname,
'../../lambda/rust-workspace'),
          containerPath: '/rust-workspace',
          },
    ],
```

Next, we will add a `commandHooks` configuration, where we can add commands that will be executed during the docker build. We will use the `beforeBundling` hook to copy the files from our rust workspace to the `node_modules` directory of the build output:

```
  commandHooks: {
          beforeInstall(): string[] {
              return [];
          },
          beforeBundling(inputDir: string, outputDir: string):
string[] {
              const napiRsMomentoModuleDir = path.join(outputDir,
    'node_modules', 'napi_rs_momento_cache');
              return [
                  `mkdir -p ${napiRsMomentoModuleDir}`,
                  `cp -r
        /rust-workspace/napi_rs_momento_cache/index.js
        ${napiRsMomentoModuleDir}`,
                  `cp -r
        /rust-workspace/napi_rs_momento_cache/package.json
        ${napiRsMomentoModuleDir}`,
                  `cp -r
        /rust-workspace/napi_rs_momento_cache/*linux*.node
        ${napiRsMomentoModuleDir}`,
              ];
```

```
        },
        afterBundling(): string[] {
            return [];
        },
    },
},
```

Note that there are only 3 files we need to copy in order to make our native module work properly:
- `package.json` - node expects to find this so it can get information about the module
- `index.js` - this is the JS code generated by napi-rs that does the work of loading the native code
- The linux `.node` file (e.g. `napi_rs_momento_cache.linux-arm64-gnu.node`) - this is the compiled rust binary.

Now we should be ready to deploy to AWS Lambda! From the `infrastructure` directory, run:

```
npm run deploy
```

If all goes well you should be able to go to the Lambda service in the AWS console, and see your second Lambda there (named `DemoNodejsRustLambda`).

Running the new Rust Lambda

Before we run the new Lambda, let's use our flush/sample scripts to ensure that the Momento cache is empty, so that we can validate that the new Lambda does its job properly. From the `lambdas/nodejs` directory:

```
npm run flush
npm run sample-cache-data
```

We should see logs indicating that the city data is not available:

```
> demo-pure-nodejs-lambda@1.0.0 sample-cache-data
> tsc && node dist/sample-cache-data.js

2024-10-14T19:48:21.331Z info: Instantiating Momento CacheClient
2024-10-14T19:48:21.410Z info: San Francisco weather: undefined
2024-10-14T19:48:21.450Z info: Seattle weather: undefined
2024-10-14T19:48:21.493Z info: Portland weather: undefined
2024-10-14T19:48:21.538Z info: Austin weather: undefined
2024-10-14T19:48:21.540Z info: PureNodeJsMomentoCache closed
```

Now we can go click the "Test" button in the AWS console to run the Lambda. When it completes, we run the `sample-cache-data` script again:

```
npm run sample-cache-data

> demo-pure-nodejs-lambda@1.0.0 sample-cache-data
> tsc && node dist/sample-cache-data.js

2024-10-14T19:51:00.451Z info: Instantiating Momento CacheClient
2024-10-14T19:51:00.605Z info: San Francisco weather:
{"minTemp":304.15,"maxTemp":304.15}
2024-10-14T19:51:00.648Z info: Seattle weather:
{"minTemp":282.04,"maxTemp":284.26}
2024-10-14T19:51:00.693Z info: Portland weather:
{"minTemp":290.529,"maxTemp":290.529}
2024-10-14T19:51:00.748Z info: Austin weather:
{"minTemp":282.04,"maxTemp":287.15}
2024-10-14T19:51:00.751Z info: PureNodeJsMomentoCache closed
```

Excellent! The cache is populated. So how long did that Lambda execution take?

Code	Test	Monitor	Configuration	Aliases	Versions

✓ **Executing function: succeeded (logs ↗)**

▼ **Details**

The area below shows the last 4 KB of the execution log.

```
{
  "statusCode": 200,
  "headers": {
    "Content-Type": "application/json",
    "Access-Control-Allow-Origin": "*"
  },
  "body": "{}"
}
```

Summary

Code SHA-256 JID9bhQYU2W6LIik/ xqBudvlCJPafigwg5moNfhNm9A=	**Execution time** 1 minute ago
Request ID c334e4ec-6646-46db-b8fb-3b54158fc640	**Init duration** 238.58 ms
Duration 10442.08 ms	**Billed duration** 10443 ms
Resources configured 8192 MB	**Max memory used** 493 MB

Only 10 seconds! Compared to 84 seconds for the pure NodeJS version!

So we have achieved an 8x performance improvement, with less than 100 lines of Rust code. Not bad for a day's work!

Cost exploration

If your primary goal when considering an optimization like this is to improve performance, then hopefully we've met that goal. However, these kinds of optimizations can also yield really important cost savings.

In the example above we reduced our Lambda execution time from 84 seconds to 10 seconds, which will reduce our Lambda bill by 8x as well. But we are still using a pretty large memory size for this Lambda container (8GB, because we needed one that large to run the original Node.js version of the code in a timely fashion). If your goal is to minimize costs then it may make sense to try running the new Rust version of our Lambda on some different Lambda sizes.

We tried re-deploying this Lambda with several different values for `memorySize`. Here is a summary of their execution time and cost:

Lambda	memorySize	Execution Time(s)	Lambda Cost/s	Execution Cost
Node.js	8192	84	$0.0001067	$0.0089628
Node+Rust	8192	10	$0.0001067	$0.0010670
Node+Rust	4096	15	$0.0000533	$0.0007995
Node+Rust	2048	26	$0.0000267	$0.0006942
Node+Rust	1024	57	$0.0000133	$0.0007581
Node+Rust	512	92	$0.0000067	$0.0006164

Total AWS Lambda Cost