Generating Unique References

Here, you'll generate unique references in the application code to satisfy the technical requirement of storage systems.

WE'LL COVER THE FOLLOWING

Before you start working on the second function, there’s an important technical aspect of working with S3 that you need to consider, which will also come into play frequently when using other storage systems. S3 isn’t really an atomic database; it’s eventually consistent. That means it’s not safe to check whether a file name exists and, if not, then upload something there. You need some way of creating unique references to avoid naming conflicts. A typical solution for this would be to use some kind of unique ID generator, but then you have a problem with traceability. It’s not easy to correlate logs and traces with outputs.

Each Lambda request has a unique request ID, which is automatically printed into the logs. You can read it from the second argument of the Lambda function, using `context.awsRequestId`. This is a great candidate for unique file names or message identifiers created from a Lambda function. Using a reference based on the request ID makes it unique but also easy to correlate with a processing session. Even more importantly, if the Lambda runtime retries processing an event after an error, it will use the same request ID. So if the function dies half-way through working and Lambda spins up another container to recover from the error, it won’t generate two different files.

Using AWS resources from Lambda functions

You can use the AWS SDK in the Lambda function to talk to other AWS

resources, including S3. Normally, when using the AWS SDK, we need to

provide authentication details when instantiating the API. When using the AWS SDK inside a Lambda function, you can just instantiate the service objects without providing credentials, like the following:

```
const aws = require('aws-sdk');
const s3 = new aws.S3();
```

Line 2 to Line 3 of code/ch7/user-form/process-form.js

Lambda functions run under temporary access credentials, valid only for a few minutes, and they are already set up in the environment by the time your function starts. AWS SDK functions will pick that up automatically.

You can now use the standard S3 SDK method `putObject` to send data to a bucket. This requires at least three parameters:

- `Body` should contain the contents of the object we're uploading. This could be a binary buffer or a string. In this case, you can just serialise the whole request to a string using `JSON.stringify`.
- `Bucket` should be the name of the target bucket. In the SAM template, you passed the bucket name to the Lambda function using an environment variable, so you can use `process.env` to read it.
- `Key` is the target file identifier on S3. You should make it unique to avoid conflicts.

Using third-party libraries in Lambda code

The AWS SDK for JavaScript is already installed in a running Lambda environment, so you do not need to specifically add it as a dependency. The command `sam build` bundles dependencies with a function, so if you need to use any other third-party libraries, just install them to the project using NPM.

You could, of course, also install the AWS SDK. This would ensure that `sam build` packages a particular version, but it would also increase the size of the function package. The deployment process would become slightly slower, which would become noticeable for very simple

functions such as the ones you're deploying. On the other hand, bundling AWS SDK with a function makes it easy to guarantee that the same

version used for development and testing is also used in production. You can try fixing the version for serious work, and use the provided library for quick experiments.

SAM does not package development dependencies, so if you want to use a third-party library just for testing, install it in your Node project with the `-D` flag (development-only flag).

When using JavaScript (not other languages), you need to convert the result of an AWS SDK call to a `Promise` object, to ensure that Lambda waits for the external call to finish before continuing. All SDK functions have a method, `.promise()`, that converts the callback-based invocation into a `Promise` object. To save the whole event payload to S3, you'll need to use code similar to the following:

```
await s3.putObject({
    Bucket: bucketName,
    Key: context.awsRequestId,
    Body: JSON.stringify(event)
  }).promise();
```

Line 7 to Line 11 of code/ch7/user-form/process-form.js

Don't forget a promise

With Node.js, Lambda functions must be `async` or return a `Promise` object. If you forget to convert an SDK call into a promise and wait for it to finish, the function will exit too soon and Lambda will kill the container before the network call completes. This does not apply to other runtimes (but you may need to synchronise with network requests differently).

In the next lesson, you will learn how to pass resource references to functions using environment variables. See you there!