Sharing Behaviour

In this lesson, you'll study the sharing behaviour of Lambda functions.

WE'LL COVER THE FOLLOWING

As you start breaking down an application into Lambda functions, some of those functions will need to share behaviours. There are currently three options for that with Lambda functions:

- Common libraries
- Lambda layers
- Invoking one function from another

These options differ primarily in four aspects:

- Latency to execute shared code
- Runtime or deployment-time consistency
- Sharing across programming language runtimes
- Deployment speed and complexity

Bundling shared libraries

The first option is to extract common code into a shared programming language library and bundle it with each function. This minimises the latency to execute shared code, since it’s just a quick in-memory call. Bundling dependencies with a function makes deployment simpler, but provides only deployment-time consistency. If you deploy just one function after changing the common code, all other functions will still run with the previous version.

Common dependencies increase the package size for each function, so this is a good choice for typical programming language libraries, but may not be the best option for bundling a 100 MB native binary for a function that needs to redeploy frequently.

Shared libraries do not have any specific impact on SAM templates. The `sam build` command automatically bundles dependencies for Java, Python, Node.js, and Go. Just list the common library in the appropriate package manifest of each dependent function (for example inside the `dependencies` section of `package.json` for JavaScript code). By the time you read this, it's highly probable that SAM will have started supporting building packages for some other languages.

In order to share data files or native Linux binaries using common dependencies, you may need to create a fake dependency package for your programming language of choice.

Working with Lambda layers

The second option is to extract common code into a Lambda layer. Layers also provide deployment-time consistency, similar to bundled dependencies, but they deploy once instead of with each function.

Dependencies included using layers also minimise latency to invoke shared code, since they effectively work inside the same container or memory space as the client code. Layers are good both for programming language libraries and for shared Linux binaries, because they can be attached to various runtimes. Functions written in JavaScript and Python can use the same layer. They cannot use the same shared programming language library.

Layers count towards the total available space for a function, so you can't work around the 250 MB Lambda code package limit by attaching layers.

Invoking one function from another

The third option is to extract common code into a separate Lambda function, and then invoke it from several other Lambda functions. Similar to using a layer, this makes it faster to deploy dependent functions. Unlike layers, the common code does not compete with other things in the dependent functions for the same 250 MB, since dependent Lambda functions deploy into different

containers. Deploying as a separate function complicates the CloudFormation stacks slightly more than deploying layers. The dependent functions will need an IAM policy that allows them to talk to the common code.

The downside of invoking a different function is that the call latency is higher than with shared libraries or layers. Instead of simple memory access, invoking a Lambda function introduces network latency and a risk of network failure between the caller and the dependency. The benefit is that this is a runtime dependency. If five functions invoke the same common Lambda function and you deploy a new version of the common code, there's no need to redeploy the dependent functions as well.

Deploying a common function also makes it possible to share behaviour across programming languages. A Lambda function written in Python can easily call a Lambda function written in JavaScript, since they will use the Lambda protocol to communicate.

In Chapter 9, you saw the difference between synchronous and asynchronous Lambda events. When one Lambda function calls another, the caller can choose if they want to wait for the result (synchronous invocation) or just call the next function and ignore the outcome (asynchronous invocation). As a general rule of thumb, try structuring calls between functions to use the asynchronous method if you can. This will make the application cheaper (because the first Lambda function does not need to wait for the other), but it will also make it much easier to maintain and evolve the code. Chains of calls are much easier to maintain than loops because dependencies always go in a single direction. Loops cause circular dependencies and tight coupling between the code of two functions.

For asynchronous function calls, it's easier to put a message queue between Lambda functions instead of directly invoking one from another. This forces you to think about the process through application events instead of remote procedure calls.

Consider the typical user registration scenario, split into two with two Lambda functions to reduce security risks. The first function writes new user information to a database. The second sends a welcome email without requiring database access. Thinking with remote procedure calls, the first

function can invoke another and send a request similar to ‘send welcome email to this address’. Designing with application events, the first function can save the new user record and then just publish an event similar to ‘this user just registered; do whatever you need’. The second function can listen to user registration events and send the appropriate email.

In terms of source code, the difference between the two options is minimal. But in terms of future extensibility, it is huge. Later on, you might want to add another activity after users register, for example scheduling an introductory phone call. In the design with remote procedure calls you’d have to modify, retest and redeploy the first function to change the process. In the design with events, you just need to add another listener to the message queue. If you wanted to stop sending welcome emails later, you could just remove the listener for that specific function, without modifying or redeploying the function that writes user information to a database.

In the next lesson, you will learn how to share configuration among Lambda functions.