Using External Storage

In this lesson, you'll explore and evaluate the different external storage options you have when using Lambda functions!

WE'LL COVER THE FOLLOWING

This chapter explains how to connect a Lambda function to persistent storage such as a file system or a database. You'll also learn about passing configurations to Lambda functions, and dealing with resource access permissions.

The application you deployed in the previous chapter is stateless. It thanks the

The application you deployed in the previous chapter is stateless. It thanks the user for submitting a form, but you are not really preserving that data anywhere. Your users won't really like just being ignored.

Lambda instances have a local file system you can write to, connected to the system's temporary path. Anything stored there is only accessible to that particular container, and it will be lost once the instance is stopped. This might be useful for temporarily caching results, but not for persistent storage. You'll need to move the user data outside the container.

Cloud storage options

There are three main choices for persistent storage in the cloud:

- Network file systems
- Relational databases
- Key-value stores

Network file systems

Network file systems are generally not a good choice for Lambda functions for two reasons. The first is that attaching an external file system volume takes a significant amount of time. Anything that slows down initialisation is a big issue with automatic scaling, because it can amplify problems with cold starts and request latency. The second issue is that very few network storage systems can cope with potentially thousands of concurrent users, so you'd have to severely limit concurrency for Lambda functions to use network file systems without overloading them. The most popular external file storage on AWS is the Elastic Block Store (EBS), which could not even be attached to Lambda functions at the time this was written.

Relational databases

Relational databases are good when you need to store data for flexible queries, but you pay for that flexibility with higher operational costs. Most relational database types are designed for persistent connections and introduce an initial handshake between the database service and user code to establish a connection. This initialisation can create problems with latency and cold starts, similar to what happens with network file systems. Since December 2019, AWS has offered a service called RDS Proxy that somewhat

reduces the problem of database connection initialisations (think of it as a managed external database connection pool). However, at the time this was written, the service was still offered only as a preview, and was only available in a small minority of AWS regions.

In general with relational databases, you have to plan for capacity and reserve it up front, which is the complete opposite of request-based pricing for Lambda functions. Increasing capacity usually involves data migration to a different storage, or stopping and restarting database server clusters. Both those operations are too slow to track auto-scaling Lambda functions. AWS now offers some relational databases on a pay-per-connection basis (for example AWS Aurora Serverless), but supporting a very high number of concurrent requests usually requires a lot of processing power, so relational databases get quite expensive.

Key-value stores

Now you are left with key-value stores as the most frequent choice for persistence for Lambda functions. Key-value stores are generally optimised for writing and retrieving objects by a primary key, not for ad-hoc queries on groups of objects. Because the data is segmented, not interlinked, key-value stores are a lot less computationally demanding than relational databases, and their work can be parallelised and scaled much more easily. AWS offers several types of a key-value stores that work well with Lambda.

The two major choices in this category are Simple Storage Service (S3) and DynamoDB. Both require no initialisation handshakes to establish a connection, they can scale on-demand so Lambda spikes will not overload them, and AWS charges actual utilisation for them, priced per request. Actually, users can choose whether they want to pay for DynamoDB based on reserved capacity or on-demand. Even in reserved capacity mode, it's relatively easy to add or remove writer or reader units according to short-term traffic patterns, so you don't have to worry about running out of capacity.

> Keeping the HTTPS connection alive
>
> Although DynamoDB and S3 do not have application hand-shake overhead, Lambda talks to them using HTTPS, which has a protocol-level initialisation overhead. If your function performs several operations on

> initialisation overhead. If your function performs several operations on the same storage in quick succession, you can speed up work by reusing the HTTPS connections. The way to achieve this depends on the programming language and client library you use to connect to AWS services, but it usually involves setting the 'Keep Alive' flag when using the AWS SDK. For Node.js, you can enable this feature by setting the `AWS_NODEJS_CONNECTION_REUSE_ENABLED` environment variable to 1.

Simple Storage Service (S3)

S3 is an *object store*, designed for large binary unstructured data. It can store individual objects up to 5 TB. The objects are aggregated into *buckets*. A bucket is like a namespace or a database table, or, if you prefer a file system analogy, it is like a disk drive. Buckets are always located in a particular region. You can easily set up *cross-region replication* for faster local access or backups. However, generally, it's best if one region is the reference data source because multi-master replication with S3 is not easy to set up.

DynamoDB

DynamoDB is a *document database*, or if you like buzzwords, a NoSQL database. Although it can keep binary objects as well, it's really designed for storing structured textual (JSON) data and supporting individual items up to 400 KB. DynamoDB stores items in *tables*, which can either be in a particular region or globally replicated. DynamoDB Global Tables supports multi-master replication, so clients can write into the same table or even the same item from multiple regions at the same time, with local access latency.

Get ready to compare the two key-value store options in the next lesson.