Processing Uploads

In this lesson, you will learn about the different options available to process your uploads.

WE'LL COVER THE FOLLOWING

This chapter explains how to trigger Lambda functions after platform events such as file uploads to S3. You will also learn about the difference between synchronous and asynchronous Lambda invocations.

The stack you deployed in the previous chapter lets users upload files quickly and easily, but the application does not do anything with those files. You have several options for how to process the uploads.

Handling conversions synchronously

The first option would be to create an API endpoint backed by a Lambda function that synchronously converts image files into thumbnails. It could send the outputs back to users directly or save the output to S3 and redirect users to the location of the result. The advantage of this option is that it would be very simple.

The issue of large files

The major disadvantage is that it would not work for large files. API Gateway will stop requests that take longer than 29 seconds. Lambda functions can work for much longer, up to 15 minutes, but API Gateway does not allow long-running tasks. If you find yourself hitting this limit, instead of trying to work around it, think about solving it with a different design. Long-running HTTP requests are liable to get interrupted. The longer the request, the more likely that unreliable coffee-shop WiFi networks might restart, user devices might try to conserve battery and suspend network actions or phones might switch from mobile networks to home WiFi and change IP addresses. Don't make people wait over HTTP.

Handling conversions asynchronously

The second option would be to handle the conversion asynchronously. In a typical three-tier server app, the usual way to handle long-running tasks would be to create several API endpoints. The first endpoint would start a background task and send the task reference back to the client. The second endpoint would allow the client to check the status of a job using the reference. The client could then deal with intermittent network problems without losing access to the results. A third endpoint would allow the client to retrieve the outcome using the job reference once the task is finished. You could do something very similar with several API Gateway resources and Lambda functions, but this would be much more expensive than it needs to be. You can make the application cheaper and faster by incorporating the serverless design.

Removing the orchestration role

In the previous chapter, you removed the need for an application server to

work as a gatekeeper and gave that role to the AWS platform instead. In a similar way, you can move some orchestration tasks from a typical middle-tier application server to the platform itself. This can significantly reduce costs. For example, instead of an API endpoint that allows clients to check whether a conversion task is complete, you could just give the client a pre-signed URL to check for results on S3. With an API endpoint, you'd have to pay for an API call, a Lambda execution, and access to S3 to check for results. With the client going to S3 directly, you only pay for S3 access. The overall usage is much cheaper and there are fewer components in between, reducing end-user latency. Moving to a serverless design makes this operation both cheaper and faster.

Similarly, you do not need a separate endpoint just to start the conversion. Many AWS platform resources can notify Lambda functions of important events. For example, S3 can directly call a Lambda function once a file is uploaded or deleted. There's no need to pay for an additional API call and a Lambda execution just for the client device to notify you that an upload is complete.

The next step for your application is to create a skeleton for asynchronous processing (see figure below). You will introduce another function to act on file uploads, without the limitation of API Gateway call duration. For now, that function will just copy the file over to another bucket so you can test the whole flow. In the previous chapter, you used the upload confirmation function (step 3 in the figure below) to generate a pre-signed URL for files in the upload bucket. You can change the URL to access the files in the results bucket instead. Clients can use that URL to retrieve the results. In Chapter 10, you will make the conversion function actually produce thumbnails.

First, a new function is created to handle file conversions. There's no need to mix the source code for the form processing functions with this function, you just need to create a new function sub-directory. You should make sure your current working directory is the one containing `template.yaml` and run the following two commands:

```
mkdir image-conversion
cd image-conversion
```

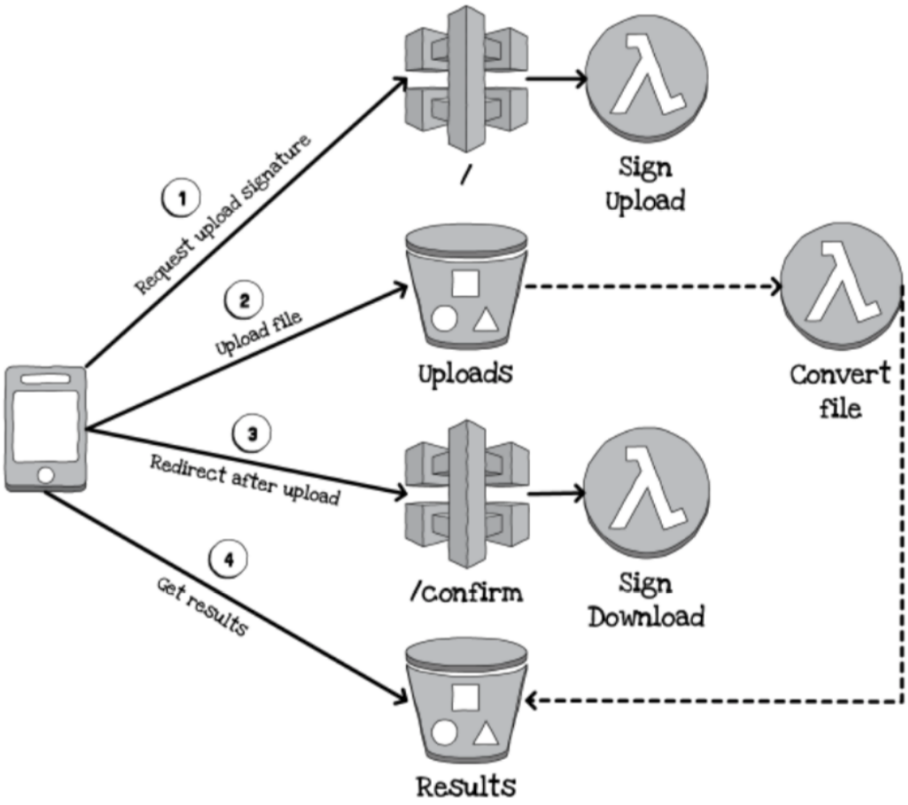
As you'll make use of the source code for this course, this has already been done

for you and you don't need to actually run the commands.

You'll use JavaScript and Node.js for this function. SAM relies on NPM package information to build functions, so you'll initialise this directory using NPM:

```
npm init --yes
```

You could have used a different runtime here because it's a different function and SAM lets you mix functions created with different languages. If JavaScript is not your preferred language, you can also find versions of the same project written in different languages at https://runningserverless.com.



Instead of an application server coordinating actions, S3 will trigger file conversion directly after upload.

Next, you'll write some code for the processing that will happen in the next lesson!