Moving Static Assets Out of Lambda Functions

Here, you'll learn how to write code to move static assets out of Lambda functions.

WE'LL COVER THE FOLLOWING

Fetching static assets

Traditional web applications bundle code and assets, and web servers are responsible for sending both to client devices. Translated to the Lambda world, that would mean including client-side JavaScript and web assets in a Lambda function, and creating at least another API endpoint and a new function to send those files to clients on demand.

Serving application contents such as images, style sheets, client-side JavaScript and HTML files through a Lambda function is a bad idea. Those files are typically public and require no authorisation. They do not change depending on individual users or requests. Paying for a Lambda function and an API call for every file request would increase operational costs significantly and introduce two unnecessary intermediaries between the user and file contents, increasing latency and degrading the user experience.

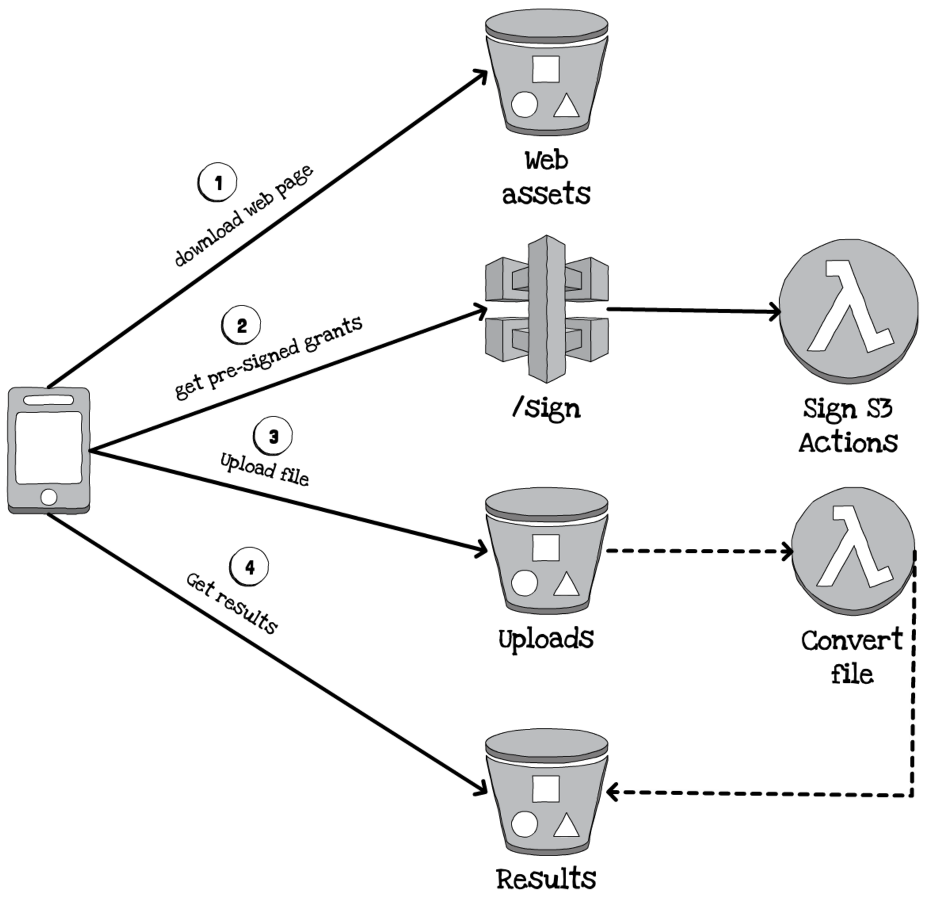
With serverless applications, it’s much more usual to put static assets somewhere for clients to fetch them directly, for example on S3. In fact, this is so common that S3 can pretend to be a web server. It even offers some basic web serving features out of the box, such as redirects and customising response headers. Browsers can directly load files from S3 using HTTPS.

In the previous chapters, you uploaded private files to S3 and generated temporary security grants for browsers to download them. You can also

upload public files, and client devices will be able to access them without a particular temporary grant. There will be no need to involve a Lambda function in the process at all. That way, you only need to pay for data transfer for static assets. This is significantly cheaper than adding another API call and a function execution. Users will also get faster responses. It will be better for everyone.

> ### Use a content distribution network
>
> For high-traffic applications, it’s usual to put a content distribution network (CDN) between users and S3 files, making the application even faster. AWS has a CDN called CloudFront, which can cache static files from S3 all over the world and even compress them automatically before sending them to users. CloudFront can also run Lambda functions to perform some quick business logic such as transforming responses or modifying incoming or outgoing headers. For your simple application, this would be unnecessary, so you’ll just use S3 directly.

Let’s restructure the application to look as in the figure given below. The client can load static assets directly from S3 without going through Lambda. A single API operation should send back both upload and download signatures, and client code can take over the remaining coordination tasks.



User workflow and session data will move to the client layer, making the application infrastructure significantly cheaper.

`user-workflow.js`

A sub-directory is created which is called `web-site` in the project directory (`app`), and inside it a file called `user-workflow.js` is created. This file will contain the whole client-side JavaScript code. For a more complex application, it would be better to split code up into individual files and somehow post-process it (for example using `webpack`), but client-side code packaging is outside the scope of this book.

First, you'll need a utility function that can retrieve an object with combined signatures for uploading and downloading, given an API URL. You'll use the HTML5 `fetch` function and parse the result as a JSON object. The code from the following listing is added to `user-workflow.js`.

```
async function getSignatures(apiUrl) {
  if (!apiUrl) {
    throw 'Please provide an API URL';
  }
  const response = await fetch(apiUrl);
  return response.json();
};
```

Line 1 to Line 7 of code/ch11/web-site/user-workflow.js

In order to upload files and handle errors gracefully in a single web page, you can't let users upload a file with a browser form submission, because that causes a page reload. You'll need a utility function to submit form data in the background using an asynchronous network request. In this case, you won't use `fetch` but the older `XMLHttpRequest` interface. Users will be uploading large files, and `fetch` does not notify callers about progress. Using `XMLHttpRequest` will allow you to visually show updates during an upload. The following function is added to `user-workflow.js`.

```
function postFormData(url, formData, progress) {
  return new Promise((resolve, reject) => {
    const request = new XMLHttpRequest();
    const sendError = (e, label) => {
      console.error(e);
      reject(label);
    };
    request.open('POST', url);
    request.upload.addEventListener('error', e =>
      sendError(e, 'upload error')
    );
    request.upload.addEventListener('timeout', e =>
      sendError(e, 'upload timeout')
    );
    request.upload.addEventListener('progress', progress);
    request.addEventListener('load', () => {
      if (request.status >= 200 && request.status < 400) {
        resolve();
      } else {
        reject(request.responseText);
      }
    });
    request.addEventListener('error', e =>
      sendError(e, 'server error')
    );
    request.addEventListener('abort', e =>
      sendError(e, 'server aborted request')
    );
    request.send(formData);
  });
};
```

Line 8 to Line 38 of code/ch11/web-site/user-workflow.js

If users try uploading a file that is too large, or if the policy expires, S3 will respond with an XML status. You'll need a small utility function to parse the XML response and get the actual error out. The following function is added to `user-workflow.js`.

```
function parseXML(xmlString, textQueryElement) {
  const parser = new DOMParser(),
    doc = parser.parseFromString(xmlString, 'application/xml'),
    element = textQueryElement && doc.querySelector(textQueryElement);
  if (!textQueryElement) {
    return doc;
  }
  return element && element.textContent;
};
```

Line 39 to Line 47 of code/ch11/web-site/user-workflow.js

The client code will need to simulate submitting a web form with a file. Browsers will let you access the contents of an input form field containing a file object as a `Blob`, and you can just post that using a standard browser `FileData` object. The following function is added to `user-workflow.js`.

```
function uploadBlob(uploadPolicy, fileBlob, progress) {
  const formData = new window.FormData();
  Object.keys(uploadPolicy.fields).forEach((key) =>
    formData.append(key, uploadPolicy.fields[key])
  );
  formData.append('file', fileBlob);
  return postFormData(uploadPolicy.url, formData, progress)
    .catch(e => {
      if (parseXML(e, 'Code') === 'EntityTooLarge') {
        throw `File ${fileBlob.name} is too big to upload.`;
      };
      throw 'server error';
    });
};
```

Line 48 to Line 61 of code/ch11/web-site/user-workflow.js

The user workflow function should periodically check whether the conversion Lambda has finished the job and let users know when the resulting file is ready to download. S3 supports partial requests by using the `Range` header. That enables you to try downloading only a small part of the resulting file (for example the first 10 bytes), to check if the file exists. The following two functions are added to `user-workflow.js`.

```
function promiseTimeout(timeout) {
  return new Promise(resolve => {
    setTimeout(resolve, timeout);
  });
};
async function pollForResult(url, timeout, times) {
```

```
async function pollForResult(url, timeout, times) {
  if (times <= 0) {
    throw 'no retries left';

  }
  await promiseTimeout(timeout);
  try {
    const response = await fetch(url, {
      method: 'GET',
      mode: 'cors',
      headers: {
        'Range': 'bytes=0-10'
      }
    });
    if (!response.ok) {
      console.log('file not ready, retrying');
      return pollForResult(url, timeout, times - 1);
    }
    return 'OK';
  } catch (e) {
    console.error('network error');
    console.error(e);
    return pollForResult(url, timeout, times - 1);
  }
};
```

Line 62 to Line 90 of code/ch11/web-site/user-workflow.js

S3 HeadObject requests

S3 also supports a `HeadObject` request that retrieves object metadata. You could have used that instead of trying to download the first 10 bytes to check for a file. That would require generating another signature in Lambda for the `HeadObject` operation, and you don't really need to care about metadata in this case, so it is simpler to just request some initial bytes to check whether a file exists.

In the next lesson, you'll look at the visual part of the workflow.