Modifying API Endpoints

In this lesson, you'll start modifying the application to make it more robust and easier to maintain!

WE'LL COVER THE FOLLOWING

- API endpoints with path parameters
  - Greedy path parameters
- `ShowFormFunction`
- `user-workflow.js`

This chapter explains how to structure SAM application templates and source code to make it easy to support, maintain, and evolve applications in the future. You will also learn about protecting serverless applications against abuse.

To avoid conflicts between user uploads, the `ShowFormFunction` function you created in Chapter 10 creates a random upload key and appends the `.jpg` image extension. In the previous chapter, you made user interactions much nicer, but you're still restricting users from converting JPG images.

ImageMagick can handle many image formats easily, so you can make the

application a lot more useful by letting users upload PNG and GIF images as well.

To support multiple formats, you just need to ensure that users upload images to S3 with the correct extensions, so ImageMagick can load the files appropriately. Restricting uploads to `.jpg` was necessary for Chapter 10 because you had no client-facing logic. `ShowFormFunction` needs to limit the uploads to something, and the client couldn't specify upfront what it wants. Now that the client code is smarter, it could ask for an upload policy for a specific image type.

API endpoints with path parameters

So far in this course, you have built API endpoints with parameters either in the query string or the request body.

There is one more option: you could put parameters in the request paths.

For example, requests to `/sign/jpg` could produce policies for a `jpg` file upload, and requests to `/sign/png` could do the same for `png` files.

You could create individual endpoints for each file type, but this would complicate the application template unnecessarily. Extending the application to support more file types would require changing the template and the application code, and it would be error-prone. API Gateway has a much simpler solution for cases when a part of the request path needs to be flexible. Declare an API endpoint using a parameter name in curly braces and API Gateway will use it for all matching requests. For example, you can create a single endpoint using the path `/sign/{extension}`, and it will handle requests for `/sign/png`, `/sign/jpg`, `/sign/gif` and any other request starting with `/sign/` in the future. To read out the value of a parameter from the path in a Lambda function, you can use the `pathParameters` field of the API Gateway event.

Any component of a path can contain a parameter, so endpoints such as `/sign/{extension}/image` are also OK. You can also add multiple generic components to the same endpoint, for example `/sign/{extension}/image/{size}`. The parameters are not completely arbitrary, though. API Gateway matches paths on non-parameterised parts and requires all endpoint paths to be unique; you can't create `/sign/{extension}` and `/sign/{type}` at the same time because API Gateway would not know how to choose between them. You'd have to add something extra to those paths to make them distinguishable. Also, only full path components can be parameters. API Gateway treats a forward slash (`/`) as a component separator. While `/sign/{extension}/image` and `/sign/{extension}/type` would be perfectly valid paths, `/sign/image.{extension}` and `/sign/{extension}-type` would not.

> ### Greedy path parameters
>
> In addition to individual path components, you can also use a *greedy parameter*, which matches one or more path components. Just include a plus sign at the end of the component name, for example `/sign/{proxy+}`. This will match `/sign/jpg` and `/sign/jpg/size/500`. This is a useful trick if you want to build your own request routing inside a Lambda function.

API Gateway does not validate path parameters, so it's not possible to set constraints directly in the API definition. For example, `/sign/{extension}` will match requests to `/sign/png`, but also to `/sign/xls`. Although ImageMagick supports lots of image types, it's still not magic enough to generate thumbnails for Excel files. To prevent user errors, you will need to validate extensions using a list of supported types in the Lambda function connected to the API endpoint.

Let's add another parameter to your application template, so you can configure the supported file types easily. The block from the following listing is added to the `Parameters` section of your application template, indented to align with other parameters (for example `ThumbnailWidth`).

```
AllowedImageExtensions:
  Type: String
  Default: jpg,jpeg,png,gif
  Description: Comma-delimited list of allowed image file extensions (lowercase)
```

Line 25 to Line 28 of code/ch12/template.yaml

ShowFormFunction

You can now change the ShowFormFunction function configuration. You'll need to update the request path to include a generic path parameter (line 11 in the following listing), and add another environment variable to pass allowed types (line 20). The rest of the function configuration stays the same.

```
ShowFormFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: user-form/
    Handler: show-form.lambdaHandler
    Runtime: nodejs12.x
    Events:
      ShowForm:
        Type: Api
        Properties:
          Path: /sign/{extension}
          Method: get
          RestApiId: !Ref WebApi
    Environment:
      Variables:
        UPLOAD_S3_BUCKET: !Ref UploadS3Bucket
        UPLOAD_LIMIT_IN_MB: !Ref UploadLimitInMb
        CORS_ORIGIN: !GetAtt WebAssetsS3Bucket.WebsiteURL
        THUMBNAILS_S3_BUCKET: !Ref ThumbnailsS3Bucket
        ALLOWED_IMAGE_EXTENSIONS: !Ref AllowedImageExtensions
    Policies:
      - S3FullAccessPolicy:
          BucketName: !Ref UploadS3Bucket
      - S3ReadPolicy:
          BucketName: !Ref ThumbnailsS3Bucket
```

Line 78 to Line 102 of code/ch12/template.yaml

user-workflow.js

On the client side, you'll need to modify the getSignatures function to add the file extension to the API call. Let's use this opportunity to make the function a bit more robust as well, and check whether the API responded with an HTTP OK (200) or not. Any application with remote network components might experience occasional transport errors. WiFi networks might drop, or users might move from one mobile network cell to another or be working on a very slow connection. Likewise, the back end might not be available to respond to user requests. AWS infrastructure is highly scalable and available, but nobody in the world is guaranteeing 100% uptime. (AWS Service Level Agreement for API Gateway is 99.95%.) You should verify the network response and show a useful error message in the event of trouble. The function in user-workflow.js useful error message in the event of trouble. The function in `user-workflow.js` is changed to match the following lines.

```
async function getSignatures(apiUrl, extension) {
  if (!apiUrl) {
    throw 'Please provide an API URL';
  }
  if (!extension) {
    throw 'Please provide an extension';
  }
  const response = await fetch(`${apiUrl}sign/${extension}`);
  if (response.ok) {
    return response.json();
  } else {
    const error = await response.text();
    throw error;
  }
};
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);
  });
};
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;
};
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';
    });
};
function promiseTimeout(timeout) {
  return new Promise(resolve => {
    setTimeout(resolve, timeout);
  });
};
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);
  }
};
function showStep(label) {
  const sections = Array.from(document.querySelectorAll('[step]'));
  sections.forEach(section => {
    if (section.getAttribute('step') === label) {
      section.style.display = '';
    } else {
      section.style.display = 'none';
    }
  });
};
function progressNotifier(progressEvent) {
  const progressElement = document.getElementById('progressbar');
  const total = progressEvent.total;
  const current = progressEvent.loaded;
  if (current && total) {
    progressElement.setAttribute('max', total);
    progressElement.setAttribute('value', current);
  }
};
async function startUpload(evt) {
  const picker = evt.target;
  const file = picker.files && picker.files[0];
  const apiUrl = document.getElementById('apiurl').value;

  if (file && file.name) {
    picker.value = '';
    try {
      const extension = file.name.replace(/.+\./g, '');
      if (!extension) {
        throw `${file.name} has no extension`;
      }
      showStep('uploading');
      const signatures = await getSignatures(apiUrl, extension);
      console.log('got signatures', signatures);
      await uploadBlob(signatures.upload, file, progressNotifier);
      showStep('converting');
      await pollForResult(signatures.download, 3000, 20);
      const downloadLink = document.getElementById('resultlink');
      downloadLink.setAttribute('href', signatures.download);
      showStep('result');
    } catch (e) {
      console.error(e);
      const displayError = e.message || JSON.stringify(e);
      document.getElementById('errortext').innerHTML = displayError;
      showStep('error');
    }
  }
};
function initPage() {
  const picker = document.getElementById('picker');
  showStep('initial');
  picker.addEventListener('change', startUpload);
};
window.addEventListener('DOMContentLoaded', initPage);
```

code/ch12/web-site/user-workflow.js

You'll also need to change the main workflow function to get the extension of the proposed file upload and pass it on to `getSignatures`. The `startUpload` function is changed to match the following listing (the relevant changes are in lines 9-14).

```
async function startUpload(evt) {
  const picker = evt.target;
  const file = picker.files && picker.files[0];
  const apiUrl = document.getElementById('apiurl').value;

  if (file && file.name) {
    picker.value = '';
    try {
      const extension = file.name.replace(/.+\./g, '');
      if (!extension) {
        throw `${file.name} has no extension`;
      }
      showStep('uploading');
      const signatures = await getSignatures(apiUrl, extension);
      console.log('got signatures', signatures);
      await uploadBlob(signatures.upload, file, progressNotifier);
      showStep('converting');
      showStep('converting');
      await pollForResult(signatures.download, 3000, 20);
      const downloadLink = document.getElementById('resultlink');
      downloadLink.setAttribute('href', signatures.download);
      showStep('result');
    } catch (e) {
      console.error(e);
      const displayError = e.message || JSON.stringify(e);
      document.getElementById('errortext').innerHTML = displayError;
      showStep('error');
    }
  }
};
```

Line 118 to Line 146 of code/ch12/web-site/user-workflow.js

In the next lesson, you'll look at how to design with ports and adaptors.