Configuring API Gateway for CORS

In this lesson, you will learn how to configure API Gateway for CORS using AWS Lambda.

WE'LL COVER THE FOLLOWING

- `OPTIONS` handler

`OPTIONS` handler

The CORS request going from the web page to your API is going to be a bit more tricky. You'll need to add an `OPTIONS` handler to the API and also change the response in your function to include the correct origin.

There are two ways of configuring `OPTIONS` handlers. The first is to add a Lambda function and an API endpoint. This would make it fully flexible, so you could dynamically calculate the right origin and response headers, but it would add an API call and a Lambda execution to your costs. In cases where you don't really need a dynamic response, such as where you always have a single valid origin, you could also just configure a static response in the API Gateway. This requires no backing Lambda, so it is cheaper but less flexible.

SAM has a nice shortcut for configuring a static response. Just add a `Cors` property to the API resource, with the supported origin. Note that this uses a slightly overcomplicated syntax. API Gateway needs quotes around header values, so you'll need to wrap the website URL in quotes using CloudFormation string substitution. The API definition is changed in your application template to match the following listing (the important addition is in line 5).

```
WebApi:
  Type: AWS::Serverless::Api
  Properties:
    StageName: !Ref AppStage
    Cors: !Sub "'${WebAssetsS3Bucket.WebsiteURL}'"
```

Line 64 to Line 68 of code/ch11/template.yaml

You can set individual CORS properties such as allowed headers or caching by providing an object instead of just a simple URL. Check out the *Cors Configuration* section of the AWS SAM API resource for more information.

The API `Cors` property is enough to handle pre-flight requests, but the response from `ShowFormFunction` will also need to send the CORS origin header back to the client browser. You'll need to let the function know about the allowed origin using another environment variable, for example named `CORS_ORIGIN`.

Because you're switching from HTML over to JSON, add another file to your `user-form` function directory to format JSON responses. The file should be called `json-response.js` and contain the code from the following listing. It will append the allowed CORS origin to all responses.

```
module.exports = function jsonResponse(body) {
  return {
    statusCode: 200,
    body: JSON.stringify(body),
    headers: {
      'Content-Type': 'application/json',
      'Access-Control-Allow-Origin': process.env.CORS_ORIGIN
    }
  };
};
```

code/ch11/user-form/json-response.js

The `ShowFormFunction` code will now be modified to generate both the upload policy and the download signature and send everything back wrapped into a JSON response. To do that, the function also needs to know about the output bucket, so you'll need to configure another environment variable later, for example `THUMBNAILS_S3_BUCKET`. The `show-form.js` function code in the `user-form` directory is changed to match the following listing:

```
const jsonResponse = require('./json-response');
const aws = require('aws-sdk');
const s3 = new aws.S3();
const uploadLimitInMB = parseInt(process.env.UPLOAD_LIMIT_IN_MB);
exports.lambdaHandler = async (event, context) => {
  const key = context.awsRequestId + '.jpg',
    uploadParams = {
      Bucket: process.env.UPLOAD_S3_BUCKET,
      Expires: 600,
      Conditions: [
        ['content-length-range', 1, uploadLimitInMB * 1000000]

      ],
      Fields: {
        acl: 'private',
        key: key
      }
    },
    uploadForm = s3.createPresignedPost(uploadParams),
    downloadParams = {
      Bucket: process.env.THUMBNAILS_S3_BUCKET,
      Key: key,
      Expires: 600
    },
    downloadUrl = s3.getSignedUrl('getObject', downloadParams);
  return jsonResponse({
    upload: uploadForm,
    download: downloadUrl
  });
};
```

code/ch11/user-form/show-form.js

Lastly, you need to modify the `ShowFormFunction` configuration in the SAM template to add the new environment variables and to let it read from the output bucket. Although the function itself will never read from the output bucket, it will need permission to do so. Remember that IAM signatures can never pass on grants that the signing key does not already have. The `ShowFormFunction` template configuration is modified to match the following listing (the important changes are in lines 18, 19 and 23-24).

```
ShowFormFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: user-form/
    Handler: show-form.lambdaHandler
    Runtime: nodejs12.x
    Events:
      ShowForm:
        Type: Api
        Properties:
          Path: /
          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
    Policies:
      - S3FullAccessPolicy:
          BucketName: !Ref UploadS3Bucket
      - S3ReadPolicy:
```

```
      - S3ReadPolicy:
          BucketName: !Ref ThumbnailsS3Bucket
```

Line 69 to Line 92 of code/ch11/template.yaml

The `ConfirmUploadFunction` is now obsolete, so you can delete that resource from the application template and the related source code files.

Get ready to upload files to S3 in the next lesson!