Lambda Utility Methods

Here, you'll get familiar with Lambda utility methods and learn how to protect the application from abuse.

WE'LL COVER THE FOLLOWING

You have already used `json-response.js` to format JSON objects into something that Lambda can understand, but this method loaded the CORS origin from the process environment, which should be the responsibility of the Lambda entry point. You can remove the process variable and add it as another regular argument, which makes this method easy to test.

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

code/ch12/user-form/json-response.js

When only the Lambda entry point depends on the actual Lambda infrastructure, you can cover utility methods with focused in-memory tests. The next listing shows what a test for the `jsonResponse` function would look like using `jest` (again, for simplicity, only the basic test cases are included, but in a real project you would include many more boundary scenarios).

```
const jsonResponse = require('../json-response');
describe('jsonResponse', () => {
  it('responds with HTTP code 200', () => {
    const result = jsonResponse('body', 'origin');
    expect(result.statusCode).toEqual(200);
  });
  it('includes the CORS origin', () => {
    const result = jsonResponse('body', 'https://gojko.net');
    expect(result.headers['Access-Control-Allow-Origin'])
      .toEqual('https://gojko.net');
  });
  it('formats objects as JSON strings', () => {
    const result = jsonResponse({a: 11, b: {c: 1}});
    expect(result.body).toEqual('{"a":11,"b":{"c":1}}');
  });
  it('uses the JSON content type', () => {
    const result = jsonResponse('body', 'origin');
    expect(result.headers['Content-Type'])
      .toEqual('application/json');
  });
});
```

code/ch12/user-form/tests/json-response.test.js

You'll need another utility method to report errors back. JavaScript is specific regarding error handling, because it can pass around exceptions or strings or just plain objects as errors, so you need to ensure that all the options are converted into a string. You'll also pass a generic HTTP error code 500 back, instead of the 200 which signals success. A file called `error-response.js` is created in the `user-form` Lambda directory, with the following code.

```
module.exports = function errorResponse(body, corsOrigin) {
  return {
    statusCode: 500,
    body: String(body),
    headers: {
      'Content-Type': 'text/plain',
      'Access-Control-Allow-Origin': corsOrigin
    }
  };
};
```

ch12/user-form/error-response.js

The test cases for `errorResponse` would look similar to the tests for `jsonResponse`, so they will be omitted from this course. You can write them for homework.

Lambda entry code

The main Lambda file now just needs to connect all the dots. It should load the configuration from environment variables, parse events and variables correctly, and format the results and errors. The `show-form.js` file is changed to match the following listing.

```
const jsonResponse = require('./json-response');
const errorResponse = require('./error-response');
const RequestProcessor = require('./request-processor');
const S3PolicySigner = require('./s3-policy-signer');
exports.lambdaHandler = async (event, context) => {
  try {
    const uploadSigner = new S3PolicySigner(process.env.UPLOAD_S3_BUCKET);
    const downloadSigner = new S3PolicySigner(process.env.THUMBNAILS_S3_BUCKET);
    const requestProcessor = new RequestProcessor(
      uploadSigner,
      downloadSigner,
      parseInt(process.env.UPLOAD_LIMIT_IN_MB),
      process.env.ALLOWED_IMAGE_EXTENSIONS.split(',')
    );
    const result = requestProcessor.processRequest(
      context.awsRequestId,
      event.pathParameters.extension,
    );
    return jsonResponse(result, process.env.CORS_ORIGIN);
  } catch (e) {
    return errorResponse(e, process.env.CORS_ORIGIN);
  }
};
```

code/ch12/user-form/show-form.js

At MindMup, we usually do not write automated tests for Lambda entry points. Instead, we manually validate them during exploratory testing. The major risk for such code is bad configuration, and automated tests before deployment do not address that risk at all. A quick manual check after deployment can easily prove that everything is connected correctly. For more complex scenarios, you can add automated smoke tests that prove that the function connects adapters and request processors correctly, and include them into a gradual deployment process as explained in Chapter 5.

Rebuild, package, and deploy the template now. Then, open the web app page and try loading a PDF file. You should see an error message telling you that the app cannot create PDF templates yet (have a look at the figure given below the following widget).

Environment Variables

Key:	Value:
AWS_ACCESS_KEY_ID	Not Specified...
AWS_SECRET_ACCE...	Not Specified...
BUCKET_NAME	Not Specified...
AWS_REGION	Not Specified...


```
{
  "body": "{\"message\": \"hello world\"}",
  "resource": "/{proxy+}",
  "path": "/path/to/resource",
  "httpMethod": "POST",
  "isBase64Encoded": false,
  "queryStringParameters": {
    "foo": "bar"
  },
  "pathParameters": {
    "proxy": "/path/to/resource"
  },
  "stageVariables": {
    "baz": "qux"
  },
  "headers": {
    "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8",
    "Accept-Encoding": "gzip, deflate, sdch",
    "Accept-Language": "en-US,en;q=0.8",
    "Cache-Control": "max-age=0",
    "CloudFront-Forwarded-Proto": "https",
    "CloudFront-Is-Desktop-Viewer": "true",
    "CloudFront-Is-Mobile-Viewer": "false",
    "CloudFront-Is-SmartTV-Viewer": "false",
    "CloudFront-Is-Tablet-Viewer": "false",
    "CloudFront-Viewer-Country": "US",
    "Host": "1234567890.execute-api.us-east-1.amazonaws.com",
    "Upgrade-Insecure-Requests": "1",
    "User-Agent": "Custom User Agent String",
    "Via": "1.1 08f323deadbeefa7af34d5feb414ce27.cloudfront.net (CloudFront)",
    "X-Amz-Cf-Id": "cDehVQoZnx43VYQb9j2-nvCh-9z396Uhbp027Y2JvkCPNLmGJHqlaA==",
    "X-Forwarded-For": "127.0.0.1, 127.0.0.2",
    "X-Forwarded-Port": "443",
    "X-Forwarded-Proto": "https"
  },
  "requestContext": {
    "accountId": "123456789012",
    "resourceId": "123456",
    "stage": "prod",
    "requestId": "c6af9ac6-7b61-11e6-9a41-93e8deadbeef",
    "requestTime": "09/Apr/2015:12:34:56 +0000",
    "requestTimeEpoch": 1428582896000,
    "identity": {
      "cognitoIdentityPoolId": null,
      "accountId": null,
      "cognitoIdentityId": null,
      "caller": null,
      "accessKey": null,
      "sourceIp": "127.0.0.1",
      "cognitoAuthenticationType": null,
```

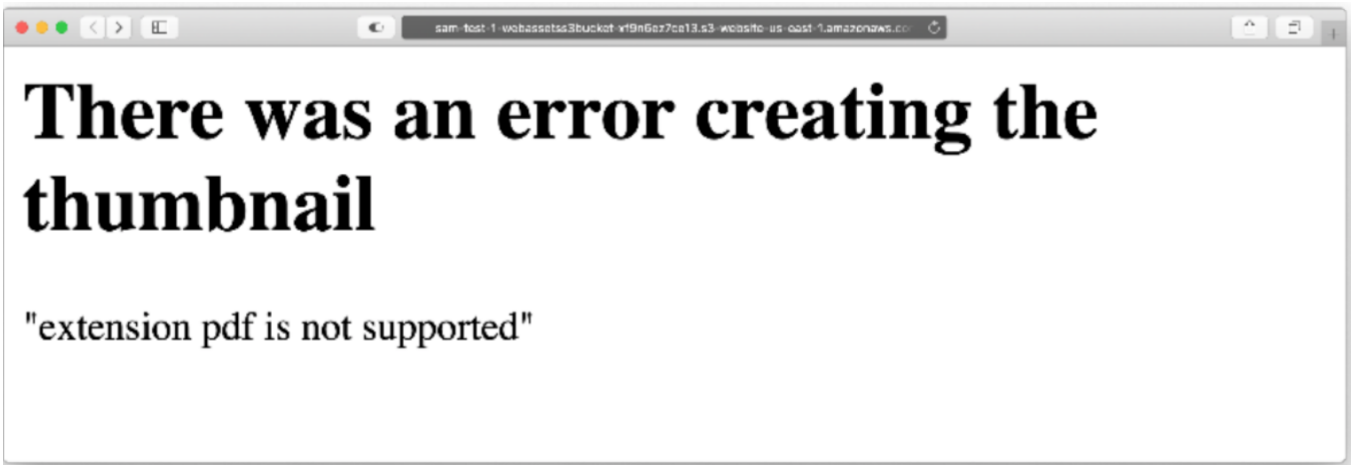
```
      "cognitoAuthenticationProvider": null,
      "userArn": null,
      "userAgent": "Custom User Agent String",

      "user": null
    },
    "path": "/prod/path/to/resource",
    "resourcePath": "/{proxy+}",
    "httpMethod": "POST",
    "apiId": "1234567890",
    "protocol": "HTTP/1.1"
  }
}
```

Protecting against abuse

So far, you have worked on making the application more robust by making it more generic and providing better error reporting.

Another important aspect of robustness is to mitigate the negative effects of abuse. Your API is currently completely exposed to the internet. A billion people coming to the app could start a billion requests, and AWS will happily scale things up to serve that. Magically autoscaling architectures that charge for usage are great when you need to satisfy increasing demand, but they can also be a financial risk. One of the biggest concerns people have when migrating to serverless applications is how to protect against someone starting a billion requests just to try to bring your system down or to cause financial damage by racking up your AWS bill. The level of this exposure, of course, depends on your risk profile, so it's difficult to provide a generic solution to this problem. However, here are some ideas that might make it easier to manage that situation.



Our application flow can now show an error message if users try to upload an unsupported file, instead of the application just timing out.

API throttling

API throttling

API Gateway can automatically throttle requests before even passing them on to the back-end integration, so one easy way of controlling the risk of abuse is to limit the number of requests coming through the API. You can configure throttling for burst requests per second or steady-state requests per second, by using the `MethodSettings` property of the API resource with SAM. You can set this in the `Globals` section for implicitly defined APIs, or in the properties of an individual `AWS::Serverless::Api` resource. For example, you can set up throttling for all the endpoints in your API by using the configuration from the following listing (the new lines are 6-10).

```
WebApi:
  Type: AWS::Serverless::Api
  Properties:
    StageName: !Ref AppStage
    Cors: !Sub "'${WebAssetsS3Bucket.WebsiteURL}'"
    MethodSettings:
      - ResourcePath: '/*'
        HttpMethod: '*'
        ThrottlingBurstLimit: 20
        ThrottlingRateLimit: 10
```

Line 68 to Line 77 of code/ch12/template.yaml

If the number of requests exceeds the limit, API Gateway will start responding with `429 Too Many Requests`. Your client application code is now more robust, so it will show the error to the client, but you could potentially change it to pause a bit and then retry automatically. For more information on API Gateway throttling, check out the page *Throttle API Requests for Better Throughput* in the API Gateway documentation.

SAM allows you to set global limits for an API stage easily, but not for individual users. API Gateway actually supports different throttling limits for different users, authenticated using API keys. You can create individual usage plans and, for example, apply a different API key for authenticated users and anonymous requests, or provide individual enterprise users with their own API keys. You then need to configure the API to require a key for invocation. For information about that, see the *Controlling Access to API Gateway APIs* page from the SAM Developer Guide.

Lambda throttling

API throttling can only protect functions invoked through API Gateway. You also have a function triggered automatically once a file is uploaded to S3, without an API in front of it. You can limit financial risk from such Lambda functions in two ways.

The first is to restrict the time allowed for execution. If someone keeps uploading massive files that require long conversions, you could tell Lambda to just interrupt those processes instead of letting them run to the end. For that, use the `Timeout` setting on the individual function. You'll need to balance this by giving the function enough time to complete the expected tasks. You're currently setting this to `600`, which is 10 minutes.

The second method to limit financial risk from Lambda functions is to control the number of concurrent executions for the function. This is effectively throttling requests on the Lambda level. By default, AWS limits concurrent executions to 1000 Lambda instances for a single AWS account (you can increase this limit by contacting AWS support). Individual Lambda functions can request a much lower limit by using the `ReservedConcurrentExecutions` property. This is a good way to limit exposure for risky functions. For example, let's allow only up to 10 instances of the conversion function to run at any given time. The code from the following listing is added to the `Properties` section of the `ConvertFileFunction` resource (indented so it is aligned with other properties, for example, `MemorySize`).

```
ReservedConcurrentExecutions: 10
```

Line 123 of code/ch12/template.yaml

For more information on concurrency controls for Lambda functions, check out the page *Managing Concurrency* in AWS Lambda documentation.

Monitoring throttling

You can use CloudWatch alerts to notify you about throttling events, in a similar way to how you used it to monitor for errors in the section *Adding deployment alerts* in Chapter 5. CloudWatch automatically tracks Lambda throttles with the `Throttles` metric inside the `AWS/Lambda` namespace. There's

no specific throttling metric for API Gateway, but a slightly wider metric

monitors all errors with codes between 400 and 499. Add an alert for the `4XXError` metric inside the `AWS/ApiGateway` namespace to learn about it.

You now have an application that is ready to receive millions of users and produce thumbnails for a variety of image formats. It does not cost anything when people are not using it, and scales automatically depending on the actual workload. Most of the operational tasks are handled by AWS, included in the price of Lambda calls. Your code is nicely structured and covered by some basic automated tests, so it will be very easy to maintain and extend in the future. You can use this code as a template for a huge number of use cases when users submit tasks and your application needs to handle them asynchronously.

In the final chapter of this course, you'll investigate how to design and structure more complex applications. You can now move to the interesting experiments in the next lesson.