Processing Request Parameters

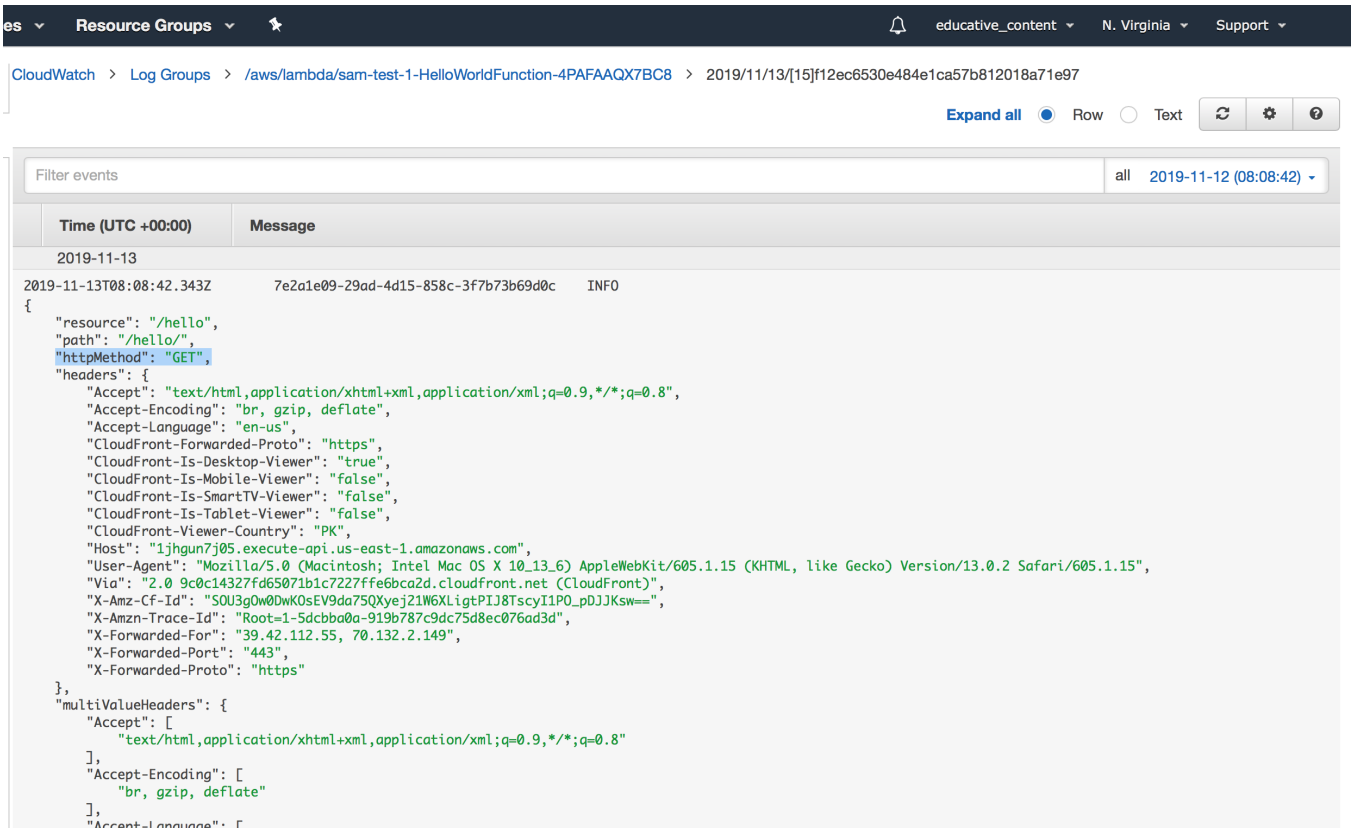
In this lesson, you will learn how to process request parameters using the `httpMethod` field.

WE'LL COVER THE FOLLOWING

You now add a handler to deal with form submissions. You did not set a specific form action URL, but you did set the form `method` to `POST`. When the form is submitted, this will make your browser send the information to the same URL where the form was displayed, but it will be using the HTTP POST method. You can use the method to differentiate between actions. If the Lambda function receives a `GET` call, it can show the form. If it receives a `POST` call, it can process the submission.

`httpMethod` field

In Chapter 4, you added some simple event logging into the sample function. Open the CloudWatch log and look at one of those events. You'll see that the method is in the `httpMethod` field of the event.



The `httpMethod` field in an API Gateway Lambda Proxy event contains the HTTP method of the request.

You can now simply set up a simple static response thanking the users for submitting the form. The function just needs to check for the `httpMethod` and decide what to send back.

```
const htmlResponse = require('./html-response');
const formHtml = `
  <html>
  <head>
    <meta charset="utf-8"/>
  </head>
  <body>
    <form method="POST">
      Please enter your name:
      <input type="text" name="name"/>
      <br/>
      <input type="submit" />
    </form>
  </body>
  </html>
`;

const thanksHtml = `
  <html>
  <head>
    <meta charset="utf-8"/>
  </head>
  <body>
    <h1>Thanks</h1>
```

```
    <p>We received your submission</p>
  </body>
  </html>
`;

exports.lambdaHandler = async (event, context) => {
  console.log(JSON.stringify(event, null, 2));

  if (event.httpMethod === 'GET') {
    return htmlResponse(formHtml);
  } else {
    return htmlResponse(thanksHtml);
  }
};
```

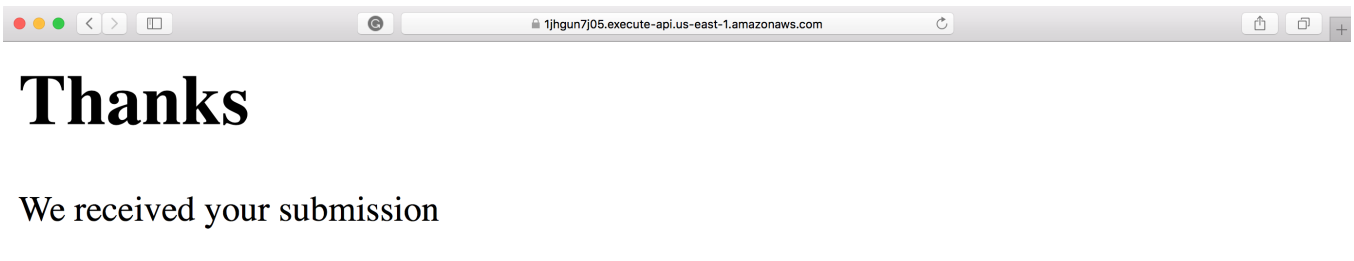
code/ch6/hello-world/app.js

In order for API Gateway to pass through POST requests, you need to add an additional event. In the template, copy the existing `HelloWorld` event and save it under a new name. Also, change its method to `post`. (Lines 14-18 are added from the following listing.)

```
HelloWorldFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: hello-world/
    Handler: app.lambdaHandler
    Runtime: nodejs12.x
    AutoPublishAlias: live
    Events:
      HelloWorld:
        Type: Api
        Properties:
          Path: /hello
          Method: get
      SubmitForm:
        Type: Api
        Properties:
          Path: /hello
          Method: post
```

Line 11 to Line 28 of code/ch6/template.yaml

Now you will build, package, and deploy the stack. Open the API URL in a browser, submit a form, and you should see a ‘thank you’ message.



The POST handler is now responding to our requests.

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"
  }
}
```

SAM hides most of the complexity of setting up an API Gateway and linking it with a Lambda function. It does, still, allow you to configure the API for advanced use cases. For some common settings, you can influence how SAM creates the implicit API Gateway by modifying the `Globals` section of the template. For more advanced settings, you need to tell SAM to skip implicitly creating an API and to use the API you configure yourself.

> ### ANY method
>
> API Gateway also supports a special marker value for HTTP methods, `any`, that matches all methods. You could have changed the original event to use `any` instead of introducing a new event here, but it's useful to see how to add events.

As an example of configuring the implicit API with a global property, you'll change the deployment endpoint configuration to optimise for regional deployment in the next section. After that, as an example of how to take complete control and skip the implicit API, you'll change the `/Prod` part of the URL to something nicer.