Avoiding Circular References

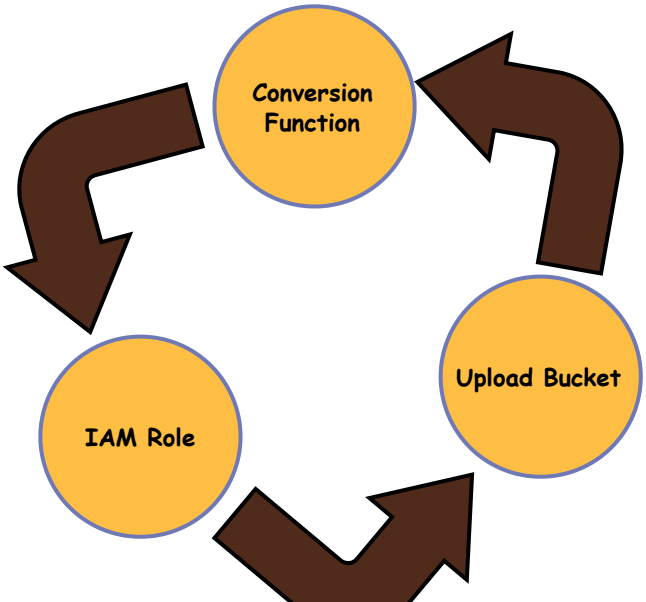
Learn how to avoid circular references by setting up a custom IAM policy.

WE'LL COVER THE FOLLOWING

- Setting custom IAM policies

The conversion function also needs permissions to read from the uploads bucket but allowing that won't be as easy as before. You can try adding another read policy into this list, but CFN Lint will complain about a circular reference. Try deploying it to CloudFormation, and you'll get the same error.

This is because SAM sets up Lambda function policies together with the IAM role for the function. To set up the function, it needs to set up the role first. To set up the role, it would need to know about the target buckets for the permissions. On the other hand, SAM sets up bucket lifecycle events, such as invoking Lambda functions, together with the bucket. So in order to set up the upload bucket, it would need to know which function reference is expecting bucket events. So the upload bucket depends on the conversion function, which depends on the role, which depends on the bucket. Hence the circular dependency.



You could, in theory, break this circle by letting the Lambda read from all the buckets, but that's a horrible thing to do from a security perspective. Instead, you'll set a custom IAM policy and not use SAM templates. This is why the uploads bucket permission was left out of the previous listing, specifying the conversion function. SAM can then create the function role, then the conversion function, then the bucket lifecycle events, and then append a policy to an existing role. That way, the function and the role will not depend on buckets during creation.

Setting custom IAM policies

CloudFormation has a resource for attaching policies to existing roles, `AWS::IAM::Policy`. You can specify an IAM policy that lets the role execute `s3:GetObject` on any resource in the target bucket, and attach it to the role after both the bucket and the role are created. The policy would depend both on the role and the bucket, but nothing would depend on the policy itself. The following lines can be added to the `Resources` section of the template (at the same indentation level as `ConvertFileFunction`).

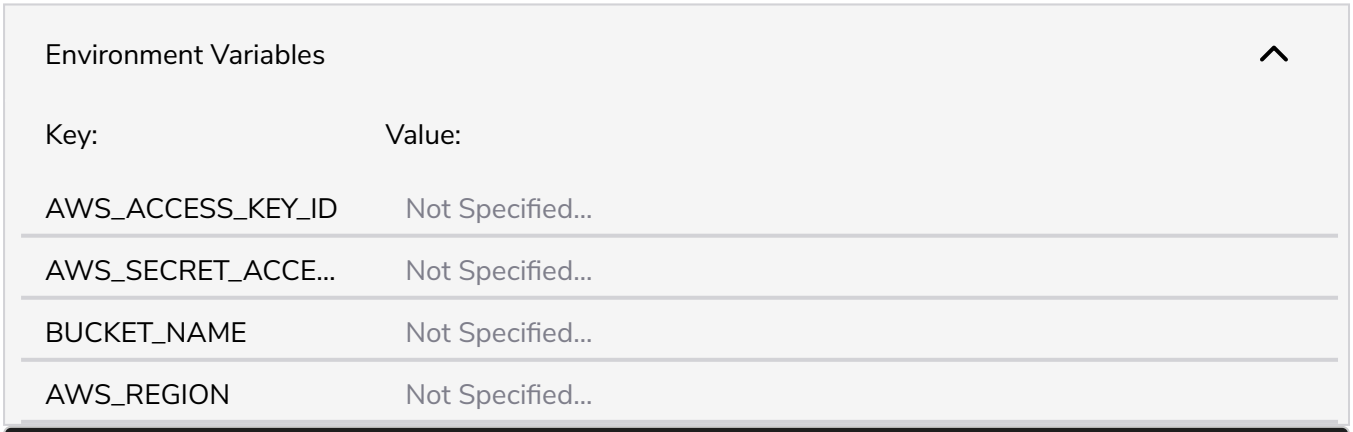
```
ConvertFunctionCanReadUploads:
  Type: AWS::IAM::Policy
  Properties:
    PolicyName: ConvertFunctionCanReadUploads
    PolicyDocument:
      Version: '2012-10-17'
      Statement:
        - Effect: Allow
          Action:
            - "s3:GetObject"
          Resource:
            - !Sub "${UploadS3Bucket.Arn}/*"
    Roles:
      - !Ref ConvertFileFunctionRole
```

Line 96 to Line 109 of code/ch9/template.yaml

With S3 policies that apply to objects, you can specify a pattern for objects, in the form of `Bucket ARN/Key`. As usual for IAM policies, you can use an asterisk (`*`) to match any value. CloudFormation S3 bucket resources have a `.Arn` property that contains the ARN of the bucket. So `!Sub "${UploadS3Bucket.Arn}/*"` (line 12) effectively means any object in the upload bucket.

Notice the reference `ConvertFileFunctionRole` on line 14. You do not have an object called that anywhere in the template, so this might seem like an error. Unless you tell SAM to use a specific role for a function, it will create a new IAM role implicitly. SAM puts the role ID into a new CloudFormation reference named after the function but with the `Role` suffix. So the implicitly created role for `ConvertFileFunction` ends up being called `ConvertFileFunctionRole`.

Now you build, package, and deploy the stack.




```
{
  "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"
  }
}
```

Play around with the webpage. You should be able to upload a JPG, PNG, or GIF image, watch the logs to know when the file is copied to S3, then click the link shown by the confirmation function. That link will display a copy of the original image from the results bucket. In the next chapter, you'll make the application resize the images in this step.

Set up a log tail for the conversion function, so you can see it in action:

```
sam logs -n ConvertFileFunction --stack-name sam-test-1 --tail
```

Of course, watching logs is not a nice user experience. You could easily adjust the client code to poll for the result using the signed URL and display a nice message while the background task is still running. There is, however, a big problem with that approach. If you try uploading a file that's not an image, you'll see it explode in the logs, but the client will have no idea what's happening. That's because S3 is using asynchronous events.

In the next lesson, you will learn how to handle asynchronous errors with dead letters.