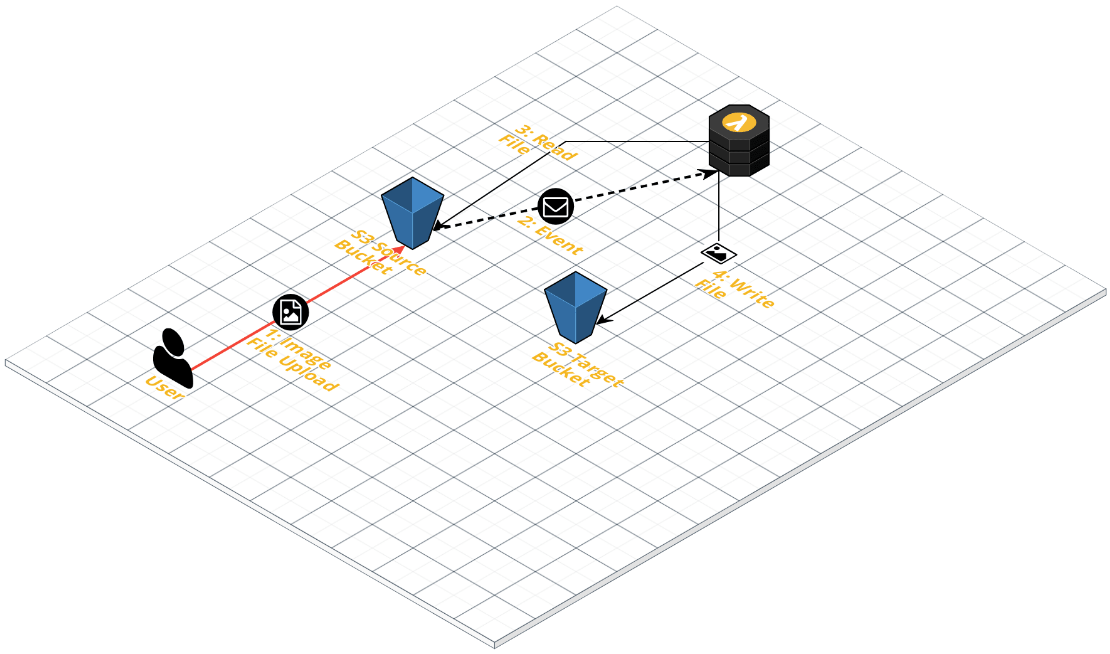
Image Manipulation

In this scenario, we will explore an **event-driven scenario** with AWS Lambda.

Lots of AWS Services generate event messages when relevant transitions take place in the environment; this transition can be a state of an VM (EC2) instance changing, but also the creation of a document in a NoSQL database or the deletion of an object in the S3 object storage facility: all these transitions generate events with payload specific to the service.

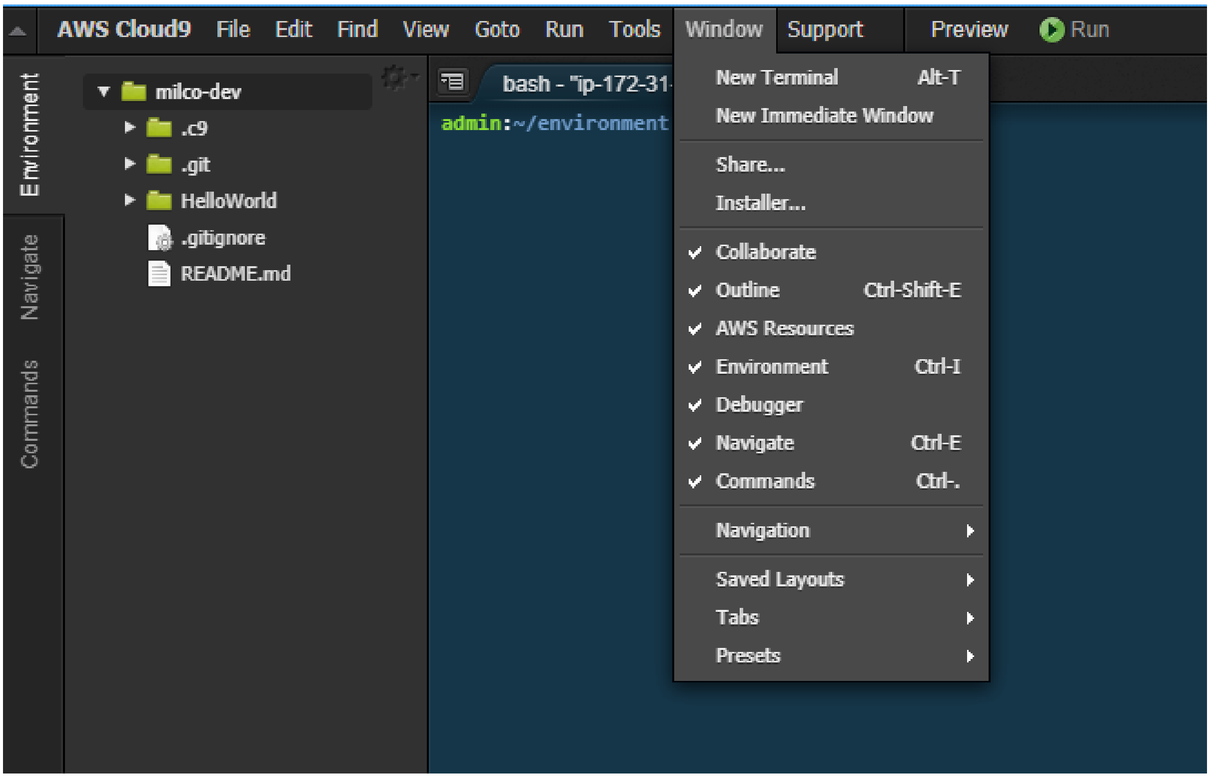
We will be creating a lambda function that will react to the creation (upload) of an image file to a specific folder in a specific S3 bucket and transform the image by applying a filter to it and write the transformed image to a different folder in the same bucket:



Relevant code snippets can be found in GitHub: https://github.com/mnuman/syntouch-aws-lambda-workshop/tree/master/3-Image%20Manipulation

Create an S3 bucket

Open a new terminal window from Cloud9:



From the terminal window, **create a new bucket using the AWS CLI.** The syntax is:

`aws s3 mb s3://<your-bucket-name>`

Note that your **bucket-name** must be *globally* unique, hence a name "test" is most probably not available ..

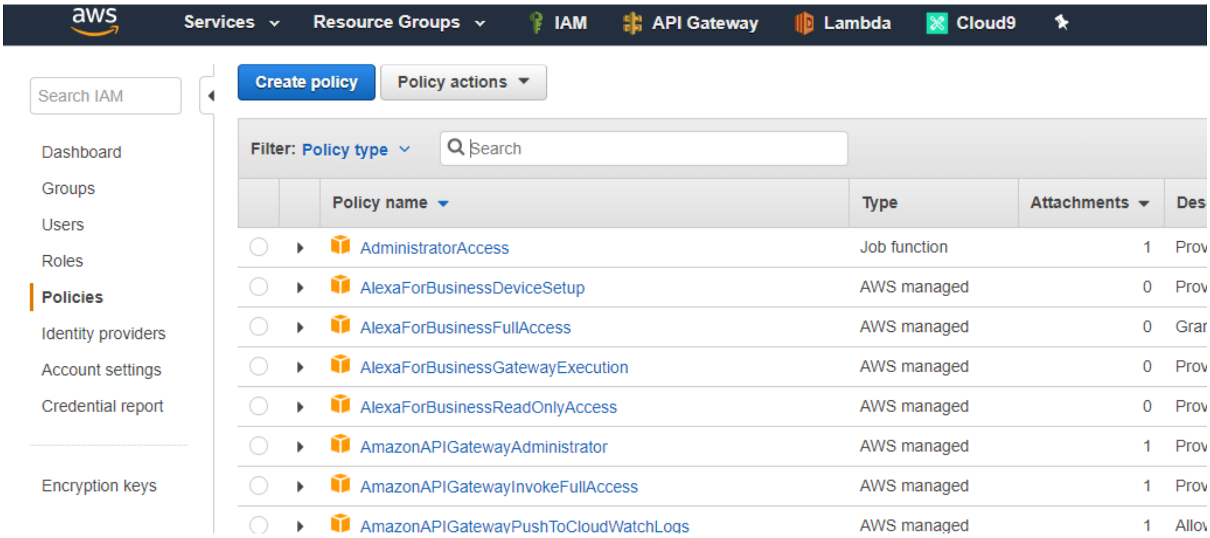
After creating the bucket **verify it exists using the aws s3 ls** command.

```
admin:~/environment (master) $ aws s3 mb s3://milco-lambda-image-bucket
make_bucket: milco-lambda-image-bucket
admin:~/environment (master) $ aws s3 ls
2018-07-04 17:27:20 cloud9-612457436284-sam-deployments-eu-west-1
2018-07-07 19:12:00 milco-lambda-image-bucket
admin:~/environment (master) $
```

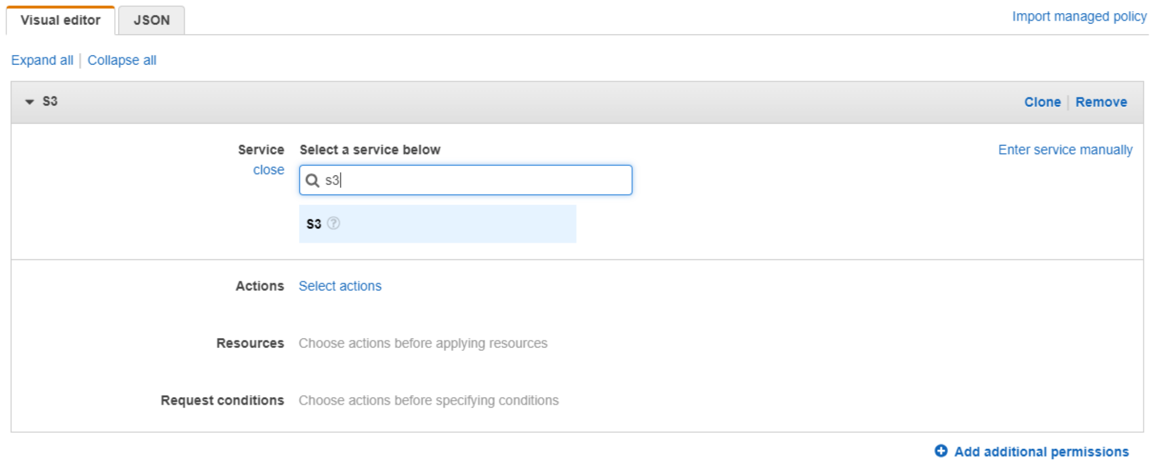
IAM Policy

Let's also create an IAM policy allowing read/write access to this bucket, so our lambda function is allowed to access it as well - apart from these permissions, the lambda function should also have the lambda basic execution role to allow it to create log events!

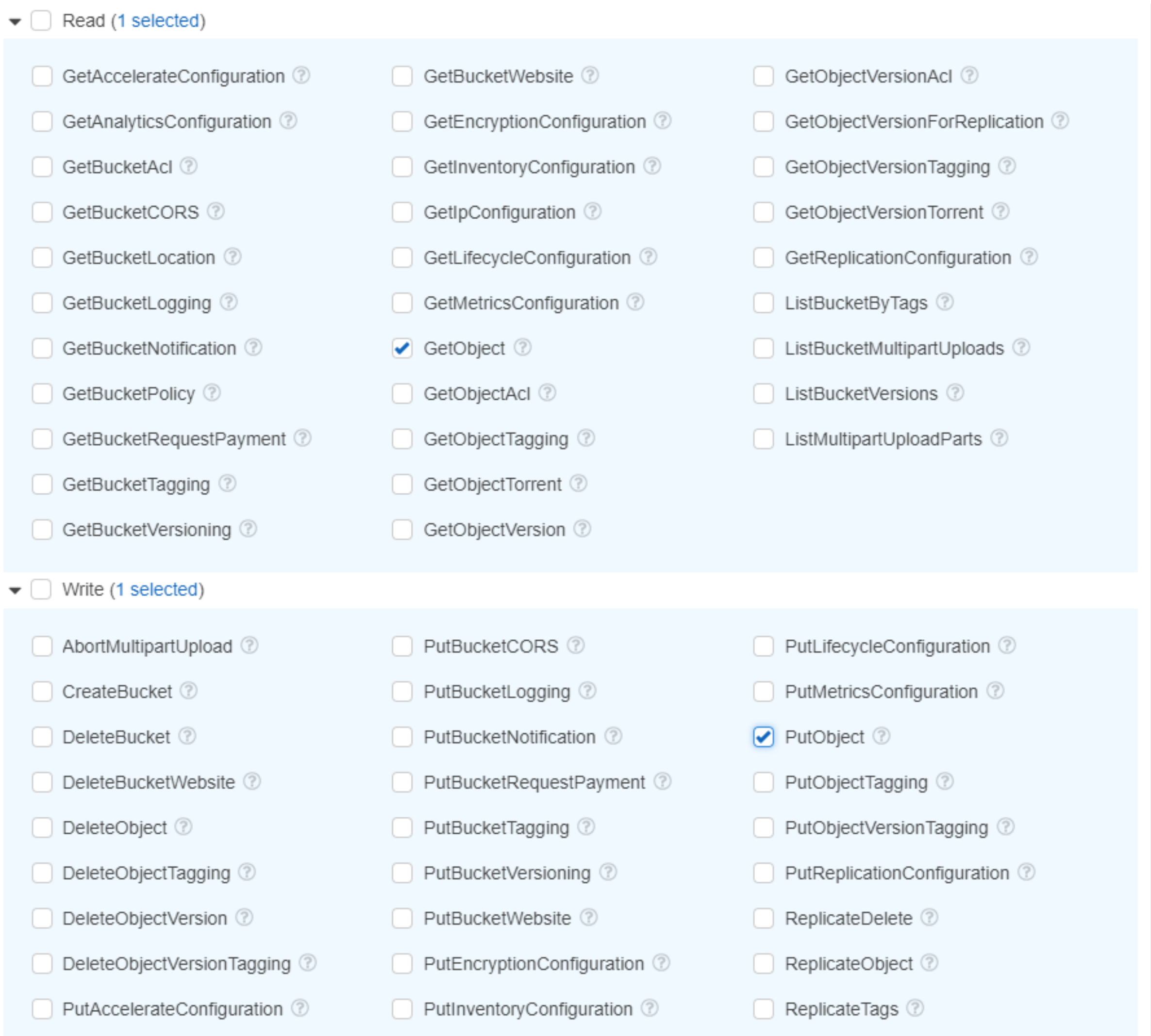
Navigate to IAM and Create Policy:



We’re building a policy to allow your lambda function access to the S3 service’s read and write operations; this authorization will be assigned to our lambda function, so it will be allowed to read the source image obtained from the event message and write the target image created by the transformation. Click “Create Policy” to start the policy creation:



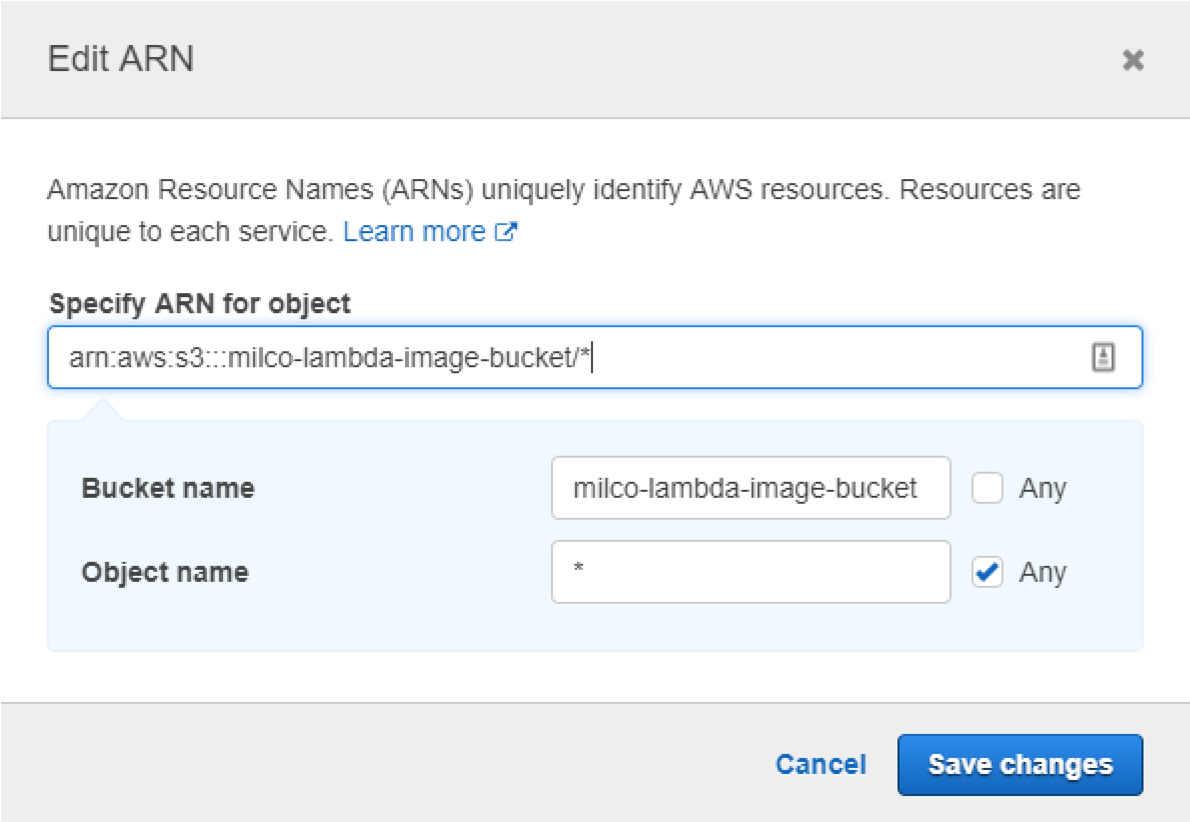
Select the GetObject from the Read section and PutObject from the Write section as the allowable actions:



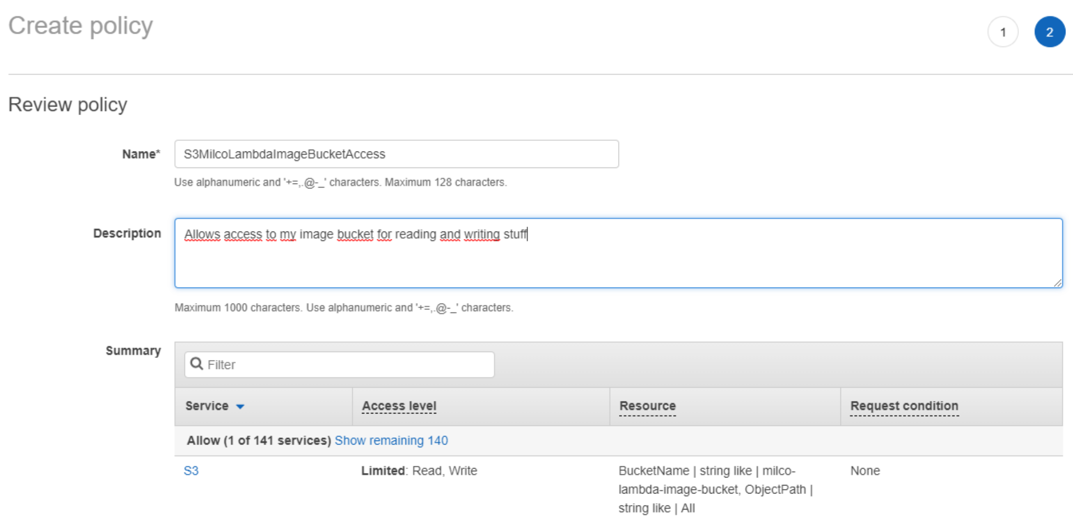
The Amazon Resource Name (ARN) is used as an unique identifier for your resources; its structure is different between services, for S3 its structure is (https://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html#arn-syntax-s3):

```
arn:aws:s3:::<bucket_name>
```

Allow the role access to your entire bucket, i.e. use a wildcard as the object name; create an ARN like the one shown (replacing your bucket's ARN):



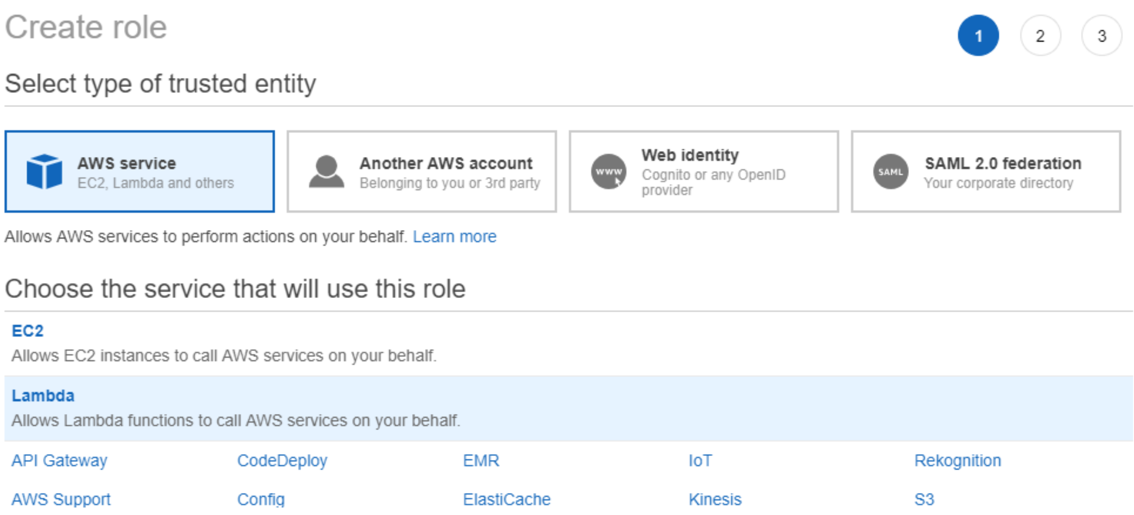
Review the policy, assign a recognizable and descriptive policy name and create it:



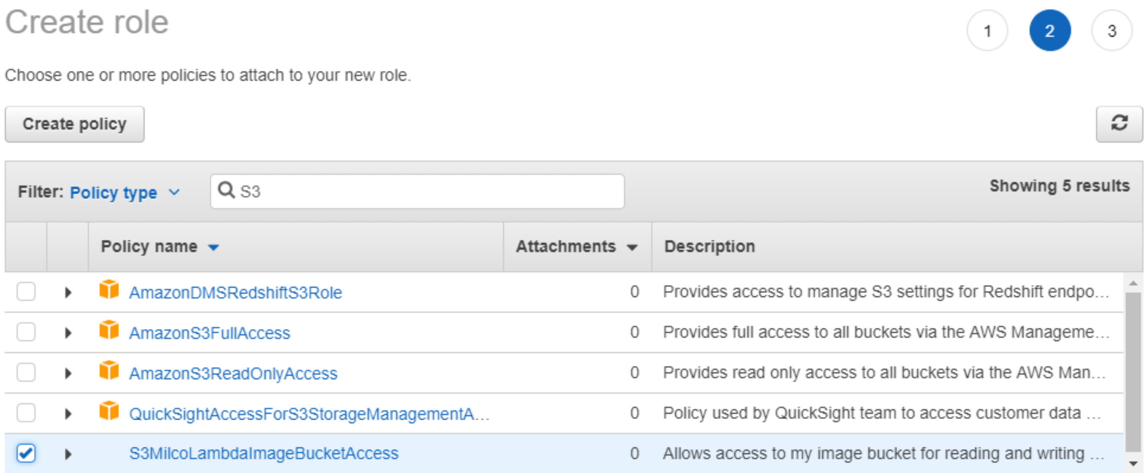
Role creation

In AWS, policies are not directly assigned to other services of AWS Lambda Functions, but instead a role is used that is assigned one or more policies; the role is then granted to the service of AWS Lambda function.

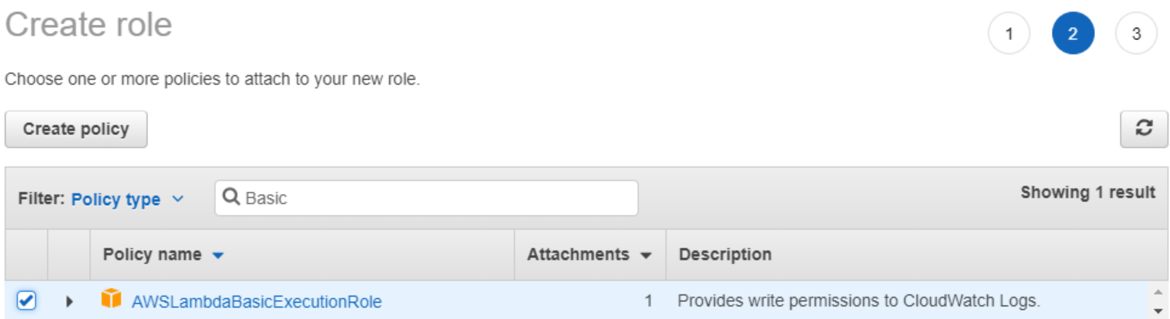
Navigate to the IAM menu, select the Roles sub item and create a new role:



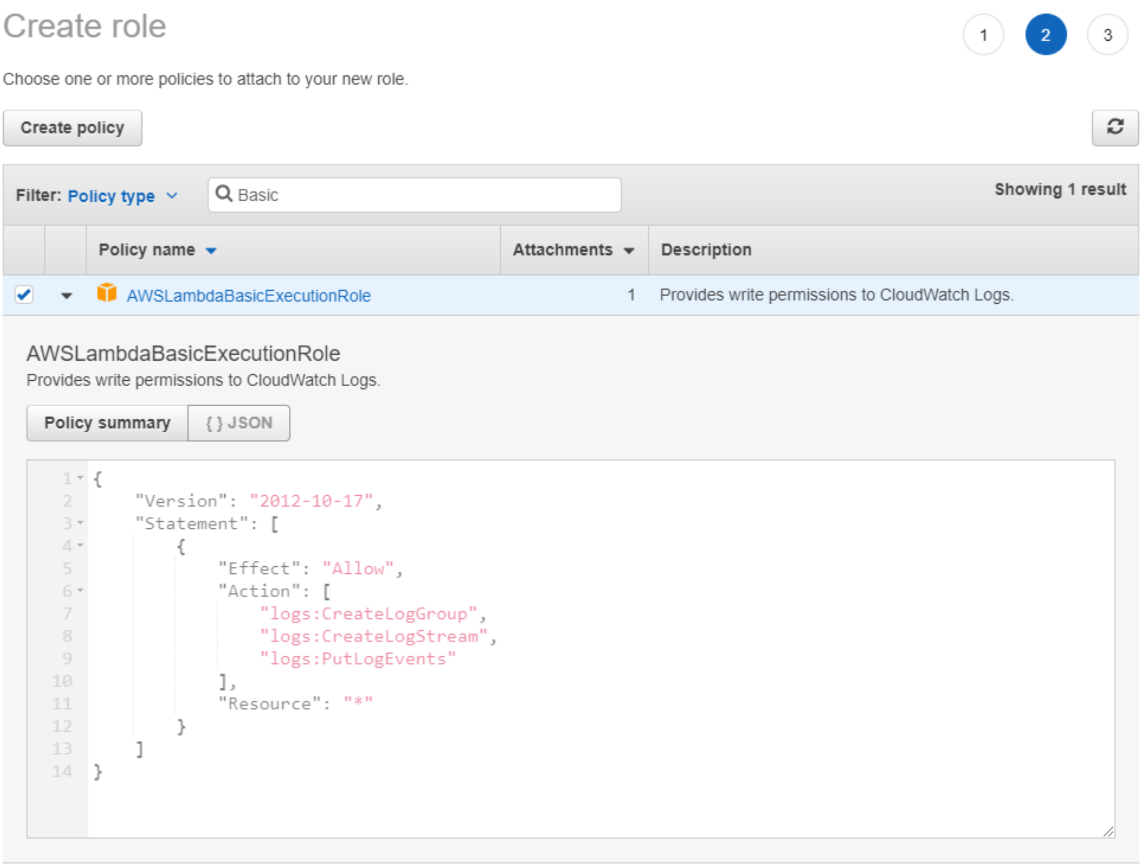
First, add the permissions defined by the policy you just created to access your bucket – in the steps before:



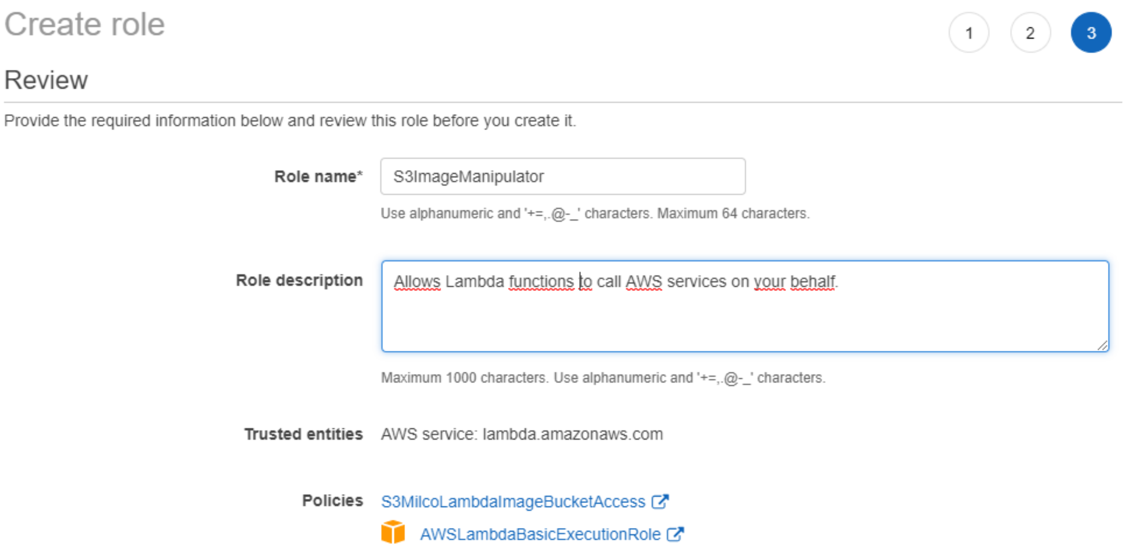
For now, **just select the box before the policy (do not press "Next" just yet)**. **Search for the AWS Basic Lambda Execution Role** to allow your function to create logging in CloudWatch as well and also select this policy:



If needed, you can expand the policy to review its contents:

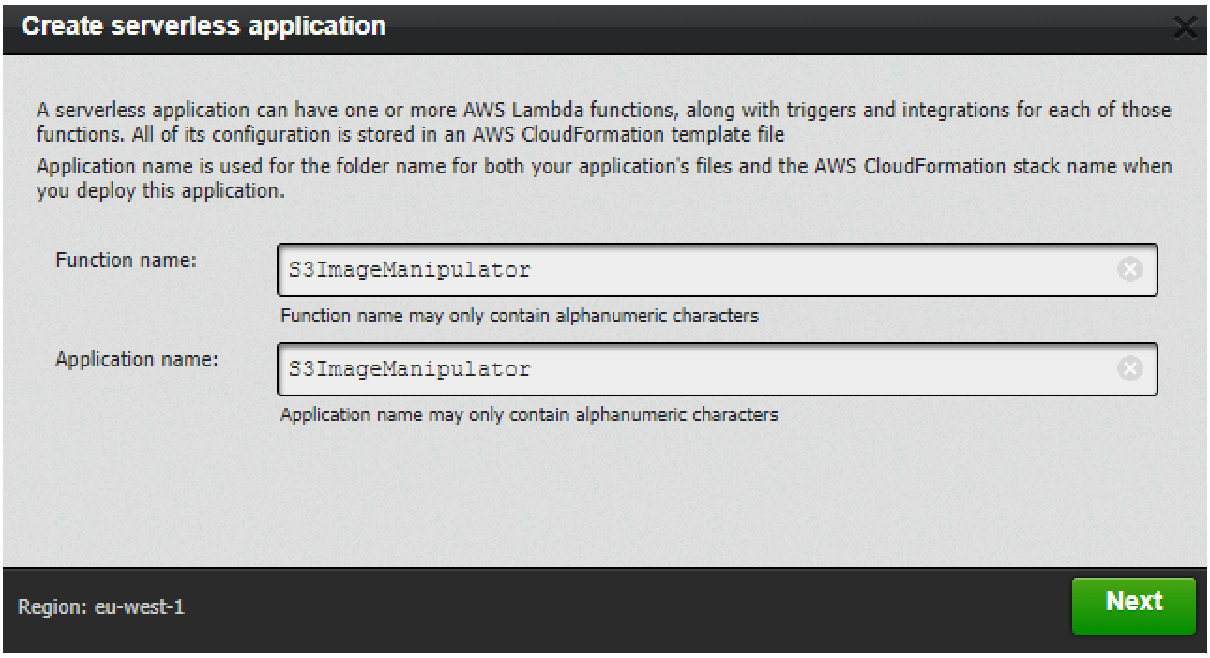


Review, assign your role name of choice and create the role:



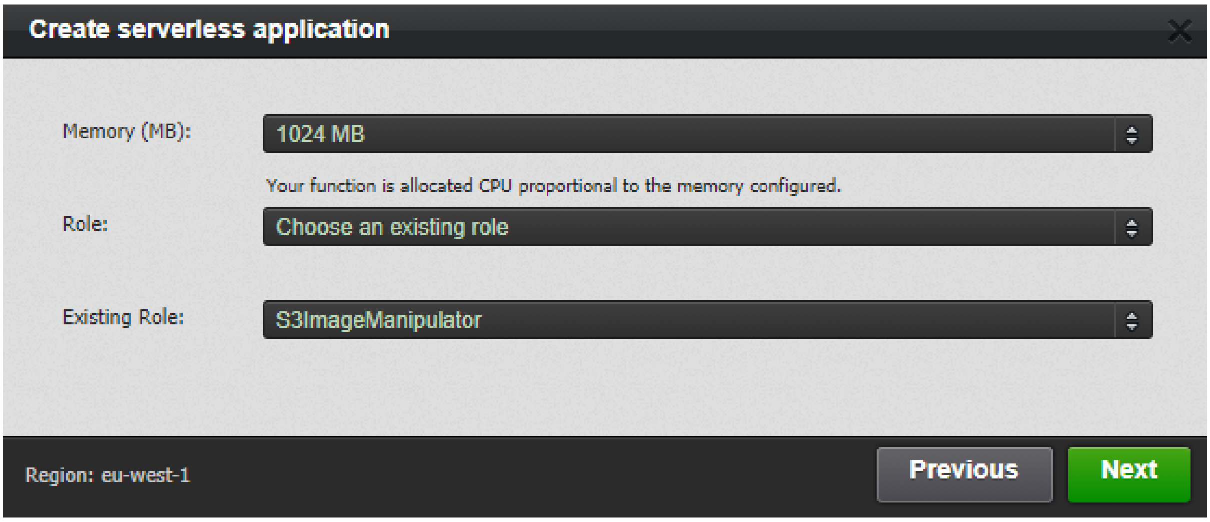
Cloud9: Application And Function

Create a new application and serverless function (in eu-west-1 = Ireland):



Select empty node.js 6.10 project and no trigger:

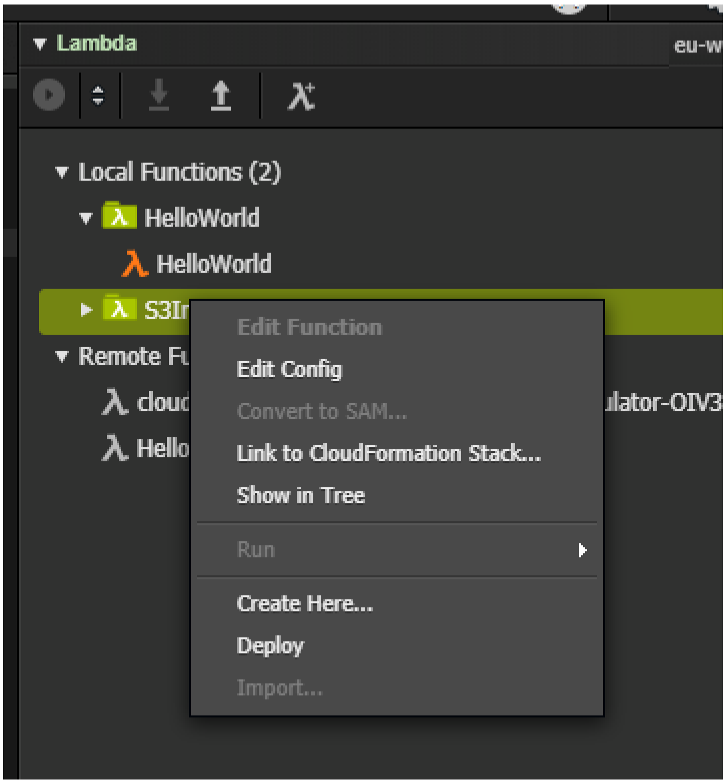
Since we're about to manipulate some images, let's make sure we have enough memory - we can adjust later - by **assigning 1024 MB**. **Select to use an existing role and select your carefully crafted execution role to allow your lambda function to actually access the bucket for reading and writing:**



In the created index.js, we **add an console.log statement** to examine the actual event payload and a **require statement to import the AWS SDK** into our node.js function for later use.

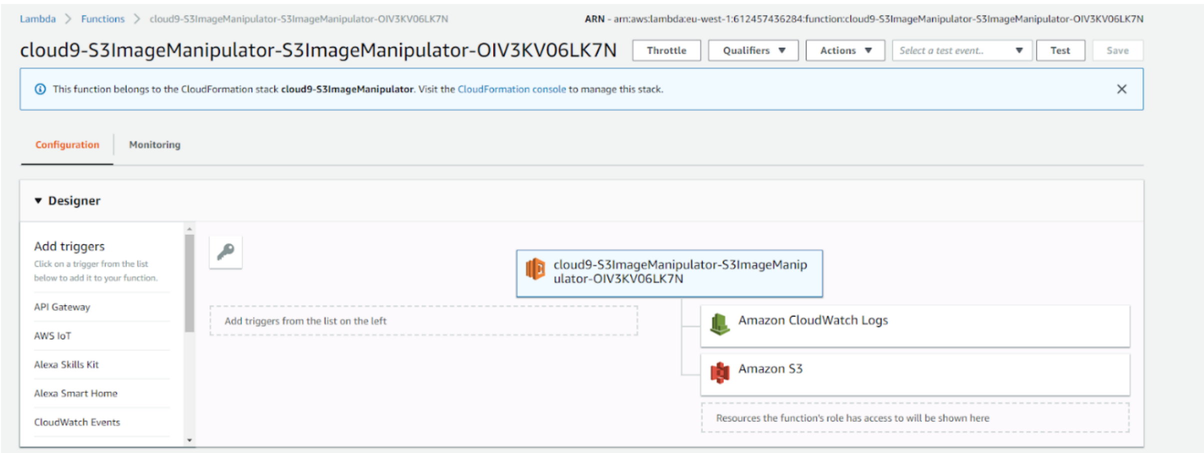
```
index.js

const AWS = require('aws-sdk');
const util = require('util');

exports.handler = (event, context, callback) => {
    console.log(util.inspect(event, { showHidden: true, depth: null }));
    callback();
};

```

Deploy the function remotely so we can link the trigger to the function for testing:

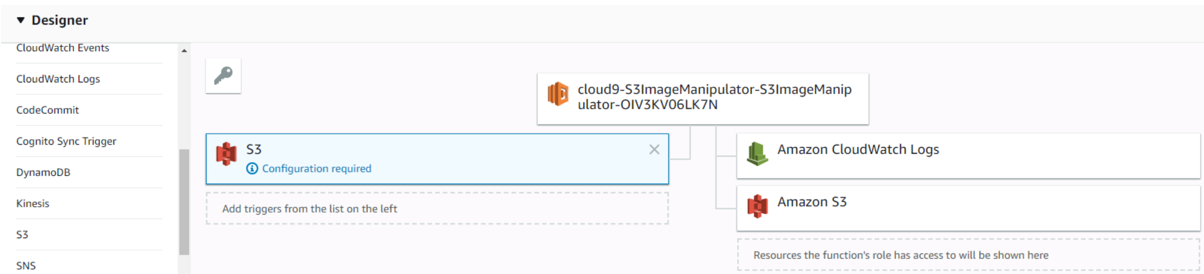


Add trigger event

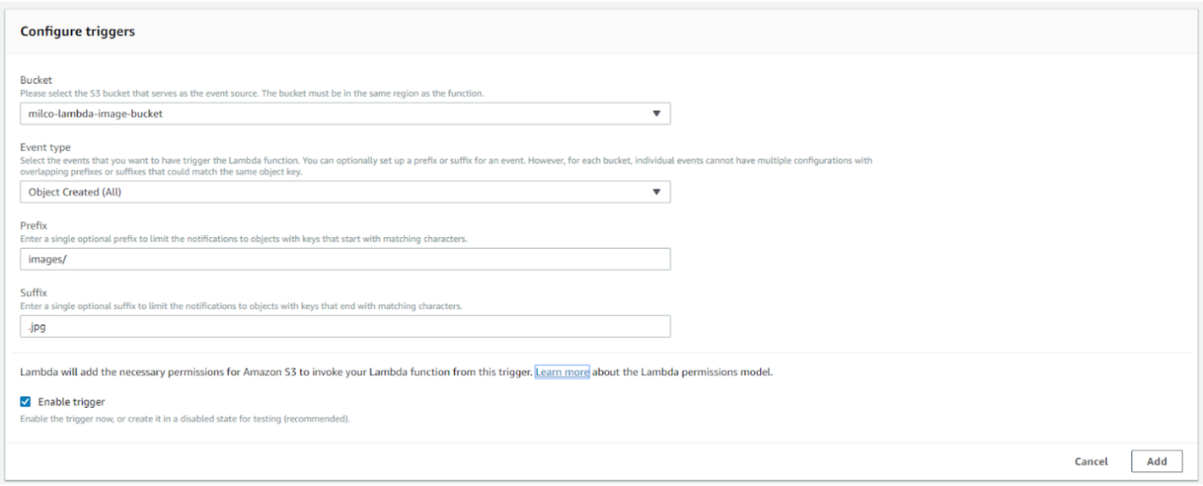
Now move over to the AWS Lambda console to attach the trigger to the function:



Lambda is showing you it requires additional configuration for the trigger to be activated:

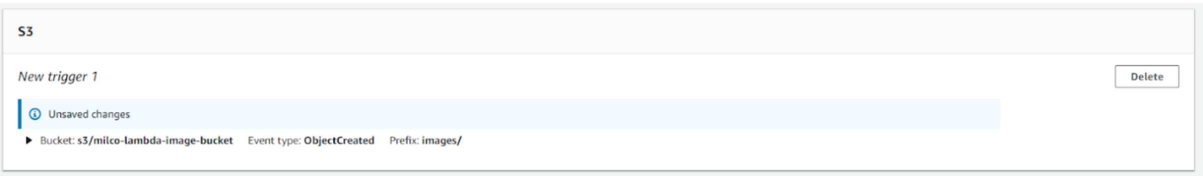


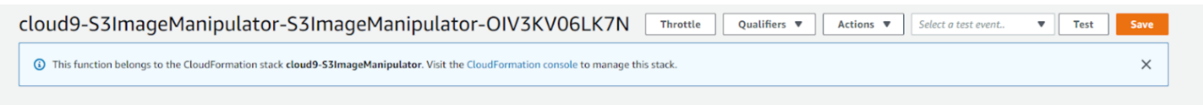
Configure the function to be triggered upon all creation events in your bucket for object that have a prefix (“folder”) of images and are of the jpg type: (this means that you should create a folder – in my case “images” within the bucket and that this trigger will only consider jpeg files – extension jpg, lowercase):



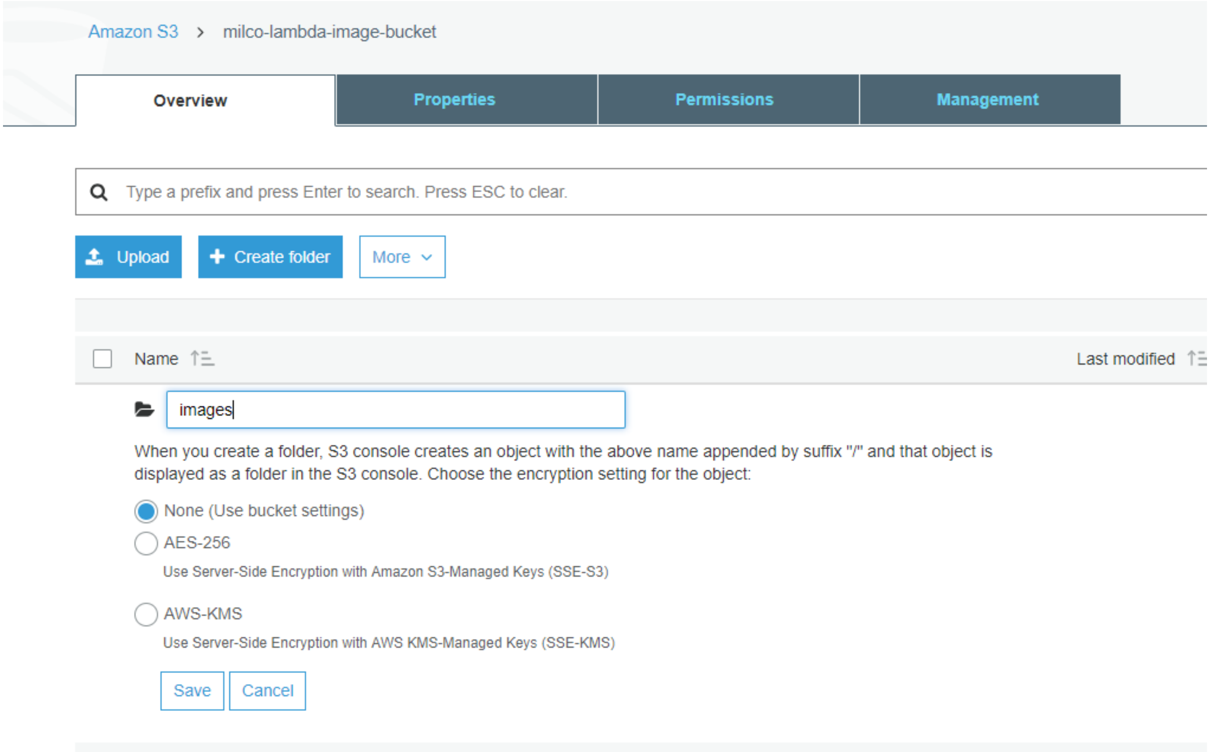
Click **Add**.

Now the trigger is defined but it still needs to be **saved**:



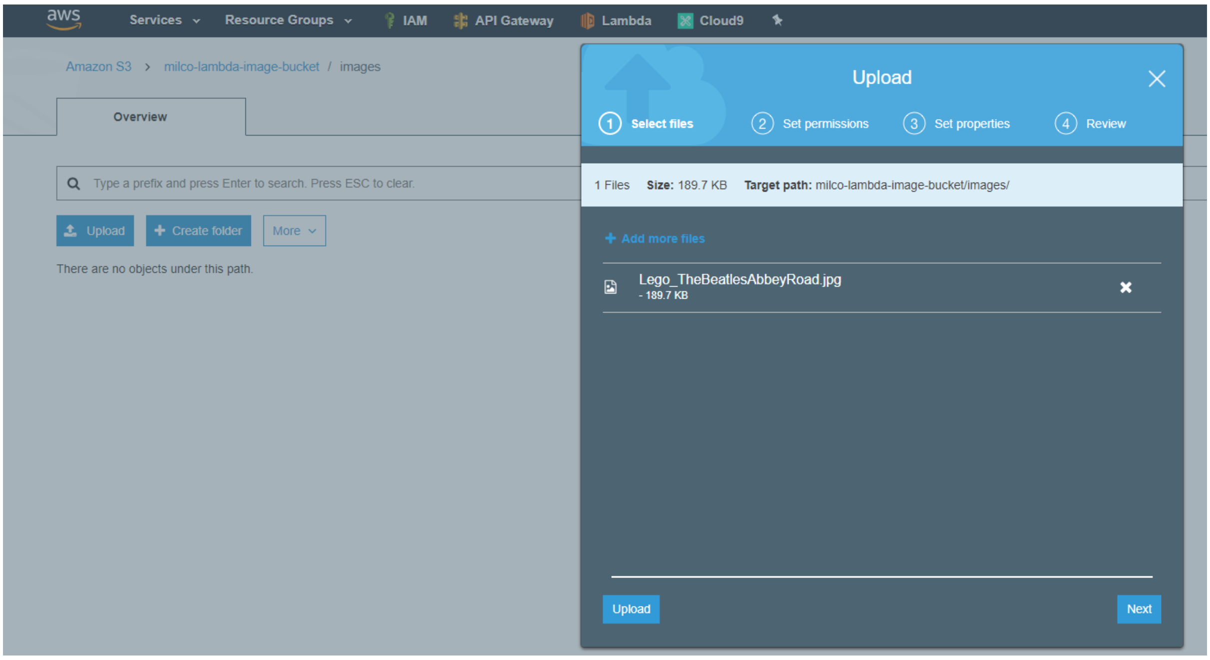


Move to the **AWS S3 console and create two folders inside your bucket**; the first must be called “**images**” as your function trigger depends on it, **you can call the destination folder “sepias”** if you intend to use the sepia transformation (other transformations suggested are grayscale or inversion – adjust if needed):

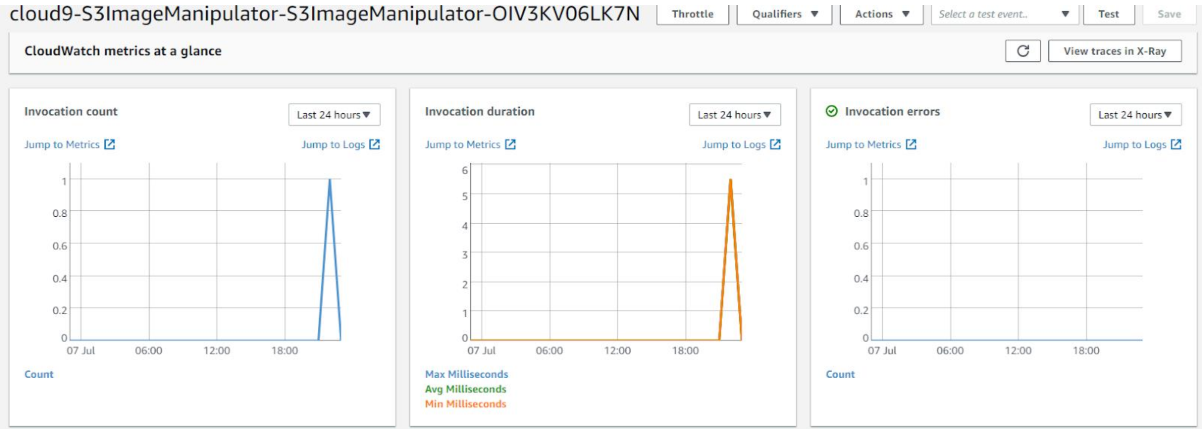


Event Structure

Move into the images prefix/folder and upload a jpg image of your choice into this folder to trigger the function and capture the event payload from cloud watch:



From the AWS Lambda Console, navigate to monitoring and verify the event has triggered the function:



Use the monitoring to go to the logs and inspect the event's structure:

Message

2018-07-07 20:29:59

No older events found at the moment. Retry.

```
START RequestId: 84a01fe9-8224-11e8-b0b2-a127bebca760 Version: $LATEST

2018-07-07T20:29:59.580Z 84a01fe9-8224-11e8-b0b2-a127bebca760 { Records:
[ { eventVersion: '2.0',
eventSource: 'aws:s3',
awsRegion: 'eu-west-1',
eventTime: '2018-07-07T20:29:58.973Z',
eventName: 'ObjectCreated:Put',
userIdentity: { principalId: 'AWS:AIDAIG7LMVO6AYW7R474I' },
requestParameters: { sourceIPAddress: '217.122.134.25' },
responseElements:
{ 'x-amz-request-id': '40A145DC3A142F6F',
'x-amz-id-2': 'NZBvk5vlx5sJuMCZyBppqVxQZwCEM1Qi+WBnJZqAhfRb2qwvTtBZv8Us00zsQm2RC7u750FWiO8=' },
s3:
{ s3SchemaVersion: '1.0',
configurationId: 'd838a270-46c4-4c87-b250-6b914e178344',
bucket:
{ name: 'milco-lambda-image-bucket',
ownerIdentity: { principalId: 'A2V2EN667TT294' },
arn: 'arn:aws:s3:::milco-lambda-image-bucket' },
object:
{ key: 'images/Lego_TheBeatlesAbbeyRoad.jpg',
size: 194284,
eTag: '7992e9c85c2ff7c94a87d860f21173bf',
sequencer: '005B4122C6E3DFAA4C' } } },
[length]: 1 ] }

END RequestId: 84a01fe9-8224-11e8-b0b2-a127bebca760

REPORT RequestId: 84a01fe9-8224-11e8-b0b2-a127bebca760 Duration: 8.04 ms Billed Duration: 100 ms Memory Size: 1024 MB Max Memory Used: 33 MB
```

Capture the actual event contents as text:

```
{ Records:
[ { eventVersion: '2.0',
eventSource: 'aws:s3',
awsRegion: 'eu-west-1',
eventTime: '2018-07-07T20:29:58.973Z',
eventName: 'ObjectCreated:Put',
userIdentity: { principalId: 'AWS:AIDAIG7LMVO6AYW7R474I' },
requestParameters: { sourceIPAddress: '217.122.134.25' },
```

```
responseElements:
{ 'x-amz-request-id': '40A145DC3A142F6F',
'x-amz-id-2':
'NZBvk5vlx5sJuMCZyBppqVxQZwCEMlQi+WBnJZqAhfRb2qwvTtBZv8Us00zsQm2RC7u750FWiO8=' },
s3:
{ s3SchemaVersion: '1.0',
configurationId: 'd838a270-46c4-4c87-b250-6b914e178344',
bucket:
{ name: 'milco-lambda-image-bucket',
ownerIdentity: { principalId: 'A2V2EN667TT294' },
arn: 'arn:aws:s3:::milco-lambda-image-bucket' },
object:
{ key: 'images/Lego_TheBeatlesAbbeyRoad.jpg',
size: 194284,
eTag: '7992e9c85c2ff7c94a87d860f21173bf',
sequencer: '005B4122C6E3DFAA4C' } } },
[length]: 1 ] }
```


```
[OMIT ABOVE FROM INSTRUCTIONS]
```

`If you want to, you can beautify the JavaScript Object through` [`http://jsbeautifier.org/`](http://jsbeautifier.org/)

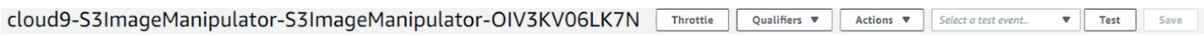
Modify the code from the AWS Lambda console to access the bucket name and the file's key from the input event; output these members to the console using console.log().

> *You can access an object using a path like expression, e.g.* `MyObject.memberfieldobject.attribute`
> *you can access elements in a list/array in Node.js by using the* `MyArray[1]` *construct, e.g.* `MyObject.MyArray[42]`
> *Note that Lists/Arrays in javascript are* ***zero-based*** *!*

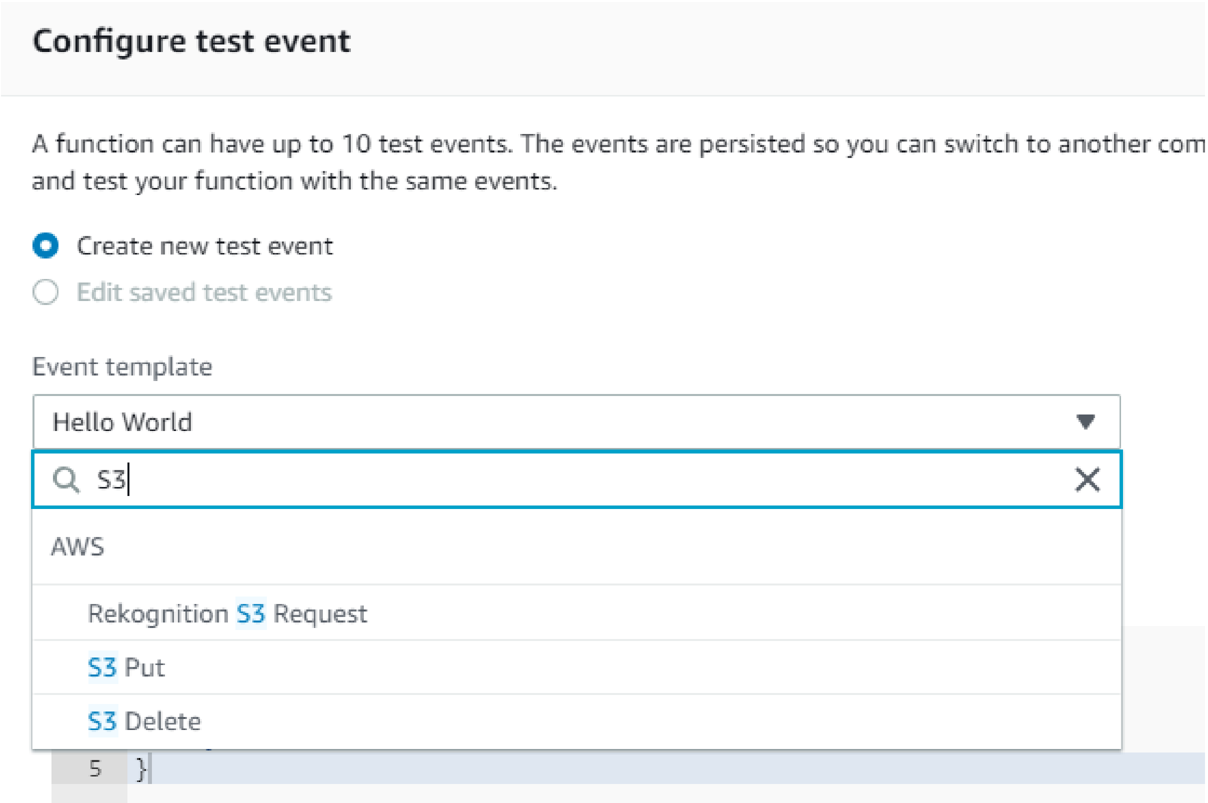
And test again …

After modifying and SAVING the code from the Lambda console, it is time to test the modification:

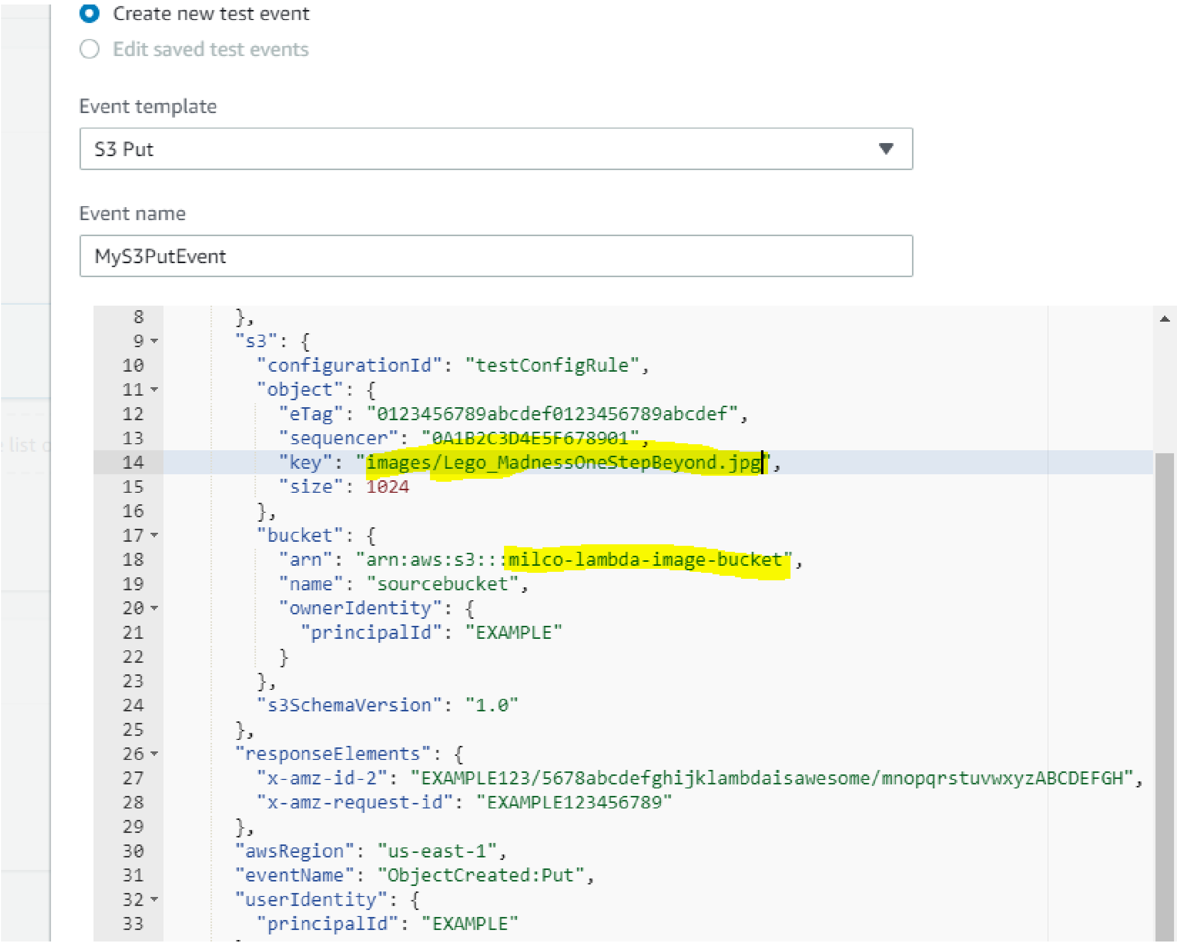
Alternatively, **select the Test option from the Lambda console**:



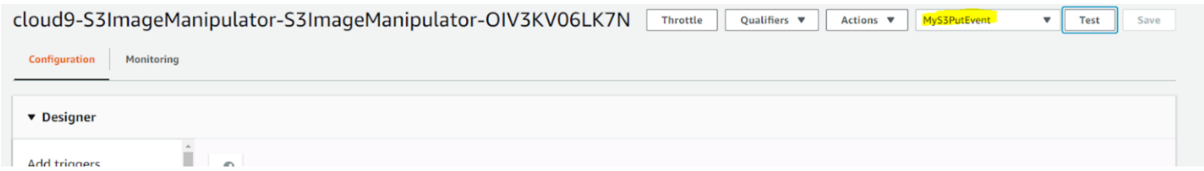
Now, **browse for S3 Put in the drop down list of standard events:**



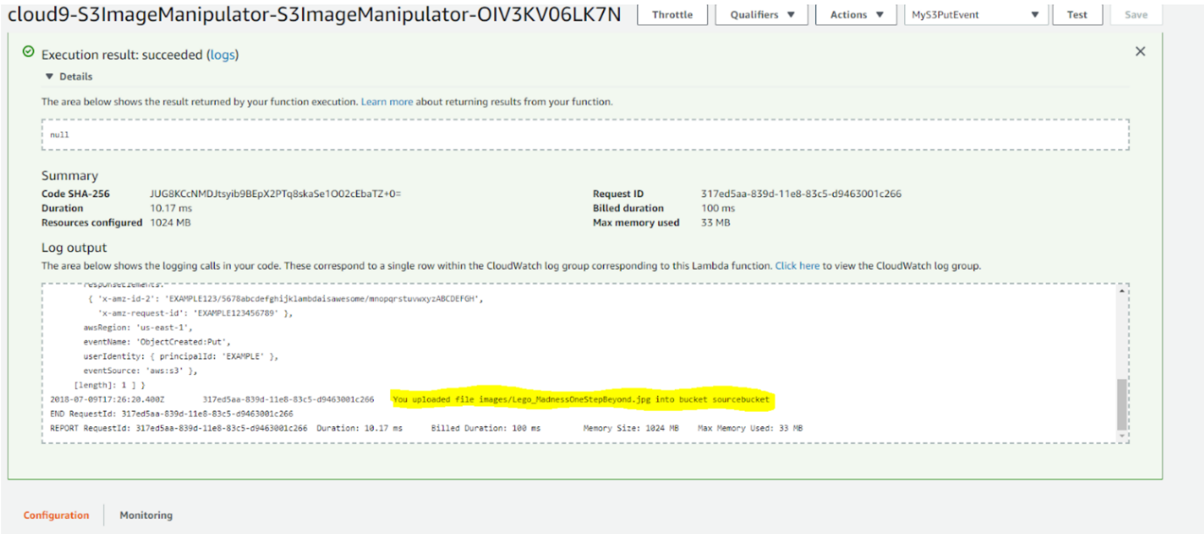
Select the S3 Put and assign a custom name to your input event; you may also want to change the bucket name and the object key to reflect a file that you can read from S3 (e.g. that was already uploaded so you may reuse the event later):



Next, kick off the function by testing with the event just created:

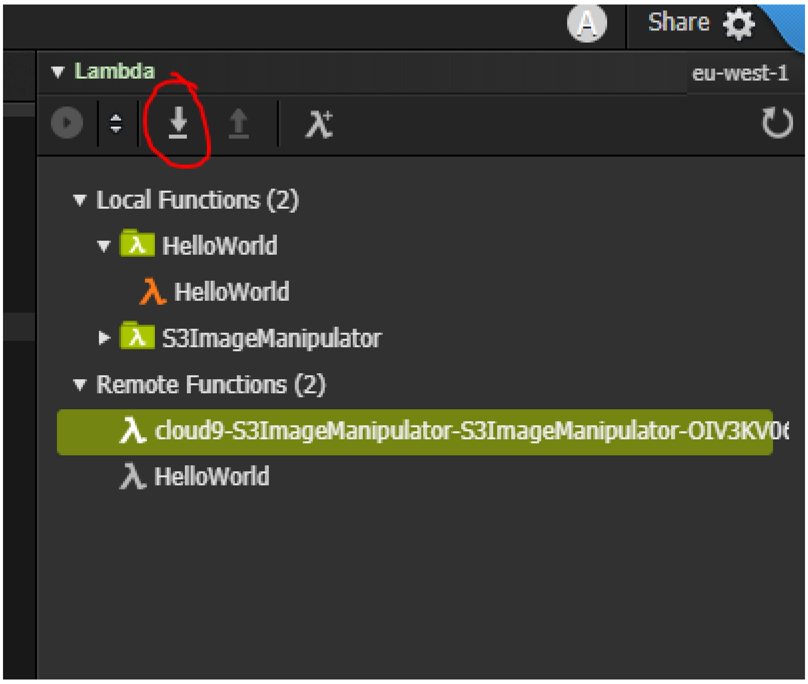


Verify that the extraction of the relevant elements for your event succeeded by verifying the output:



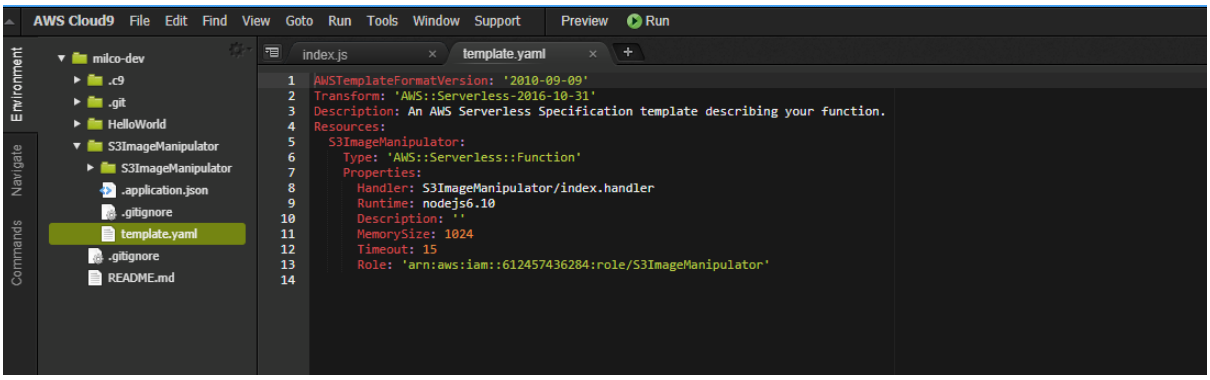
Synchronize code and config

You can retrieve the changes you made in the AWS Lambda Console into the Cloud9 environment, by **selecting the remote Lambda function and selecting the Import option in the toolbar**:

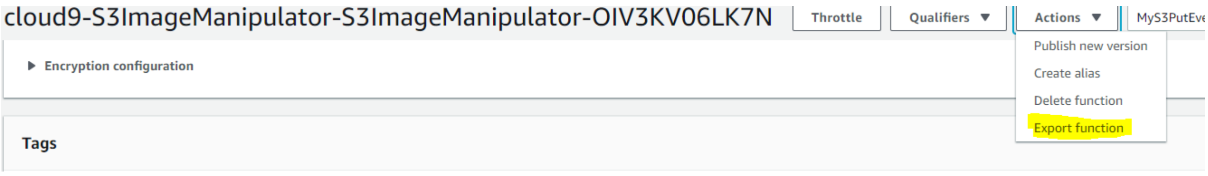


Now we roughly know how to set things up, it’s time to secure this setup as well into our environment. The actual setup is being provision by SAM (Serverless Application Model, which is an extension to CloudFormation) under the hood. This service is driven by a template.yaml in your application’s (not function’s!) top level directory to provision the stack.

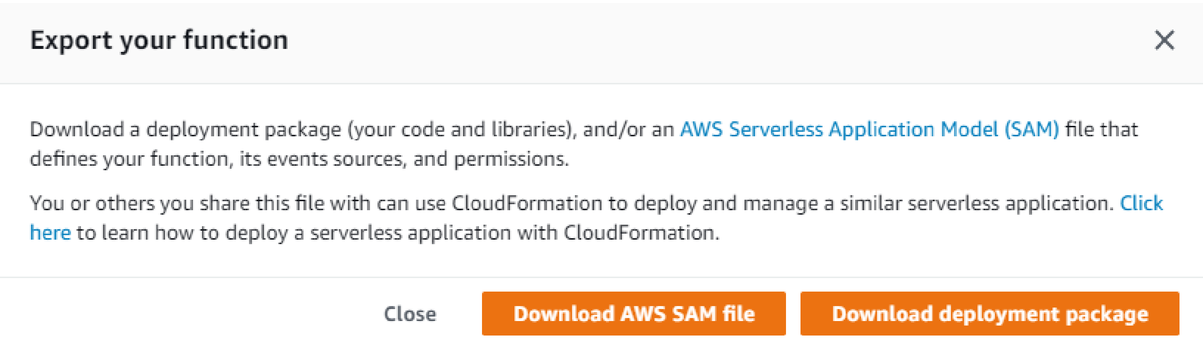
Open this file in your Cloud9 editor to make the required changes:



The easiest way to retrieve the current function setup is to simply export it from the AWS lambda Console:



When prompted, **download the SAM file**:



Open the file in your editor and replace the current contents of the template.yaml file in your Cloud9 IDE with the downloaded version's content.

The Serverless Application Model's Syntax is documented on GitHub
https://github.com/awslabs/serverless-application-model

template.yaml

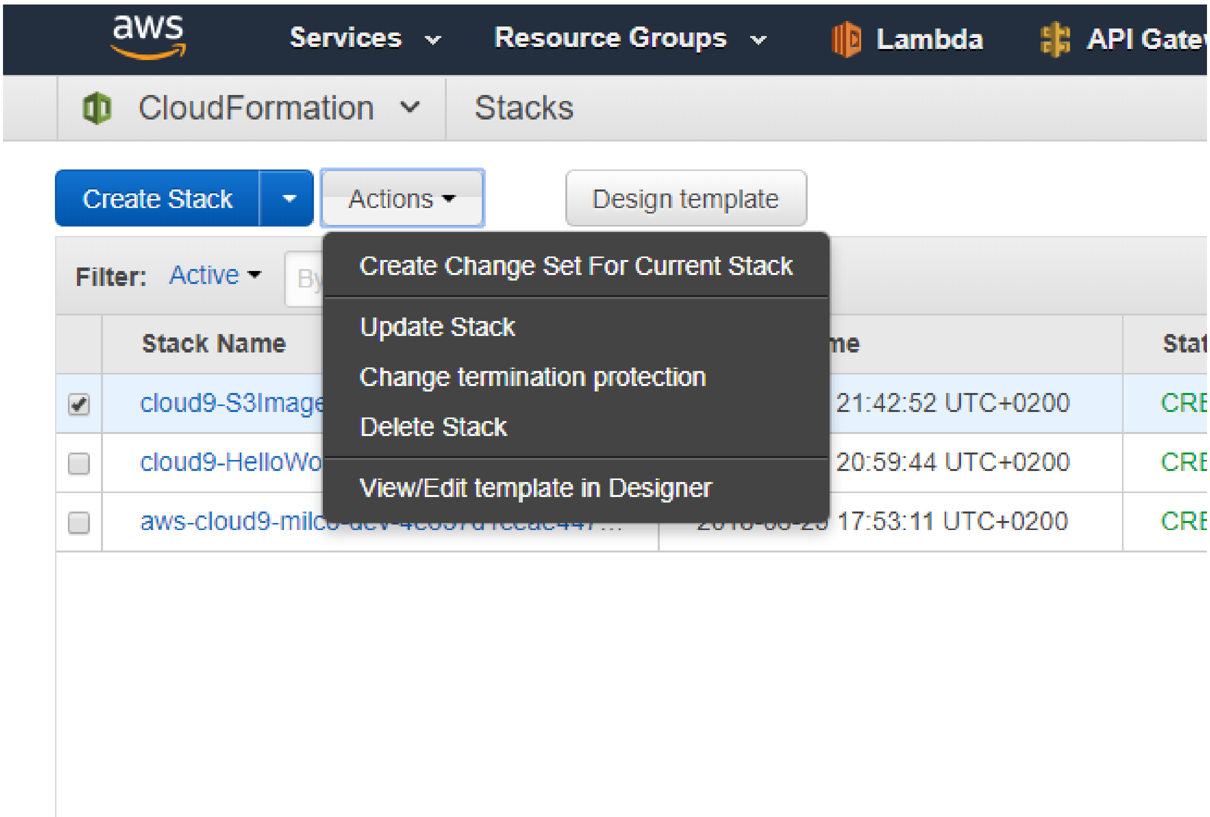
```yaml
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: This is my image manipulator
Resources:
  S3LambdaImageManipulator:
    Type: 'AWS::Serverless::Function'
    Properties:
      Handler: S3ImageManipulator/index.handler
      FunctionName: S3ImageManipulator
      Runtime: nodejs6.10
      CodeUri: .
      Description: This is my image manipulator
      MemorySize: 1024
      Timeout: 60
      Role: 'arn:aws:iam::612457436284:role/S3ImageManipulator'
      Events:
        ObjectCreatedEvent:
          Type: S3
          Properties:
            Bucket:
              Ref: MyInputBucket
            Events: s3:ObjectCreated:*
            Filter:
              S3Key:
                Rules:
                  - Name: prefix
                    Value: images/
                  - Name: suffix
                    Value: jpg
  MyInputBucket:
    Type: 'AWS::S3::Bucket'
    Properties:
      BucketName: mn-image-bucket
```

In your modified template you should make several modifications:

- **add a FunctionName for the serverless function**
- **update the timeout to a whopping 60 seconds**
- **add the triggering event according to our initial definition, i.e. triggered by the creation of an S3 object in our bucket (handsomely renamed), prefixed by images/ as the folder name and of type jpg**.

Clean up yourself - your mama doesn't work here!

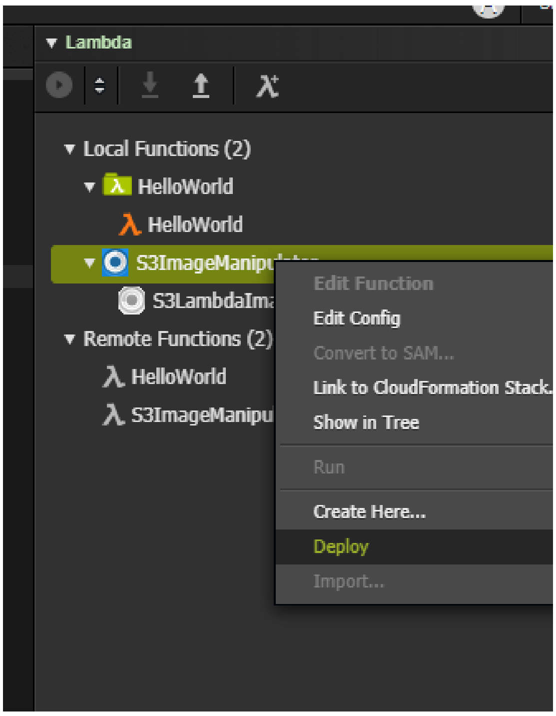
Before (re-)creating the stack, clean up the stack based on the OLD definition. The SAM stacks are provisioned by AWS CloudFormation, so **navigate to CloudFormation** in the Services drop-down and **delete the stack you created** earlier - the name should reflect which stack to select …



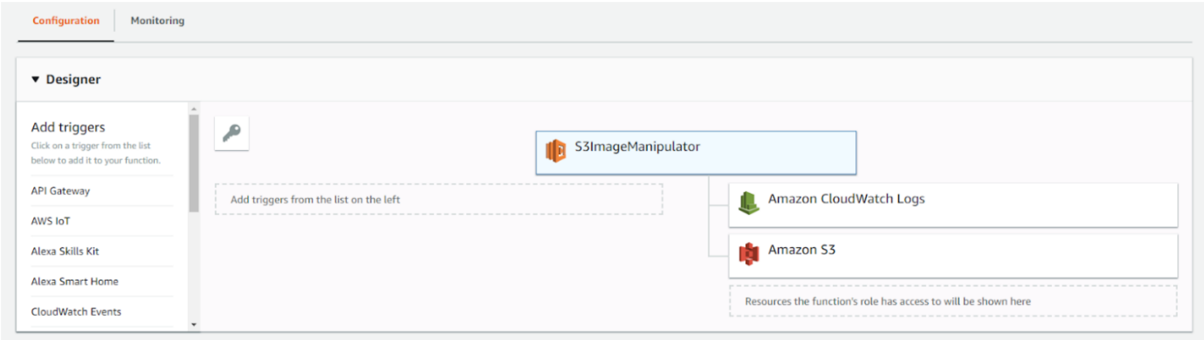
It will take a minute or so to delete the stack; the deletion may fail on objects being present in your bucket - if so, navigate to S3, delete the objects and try again!

Deploy your own new stack

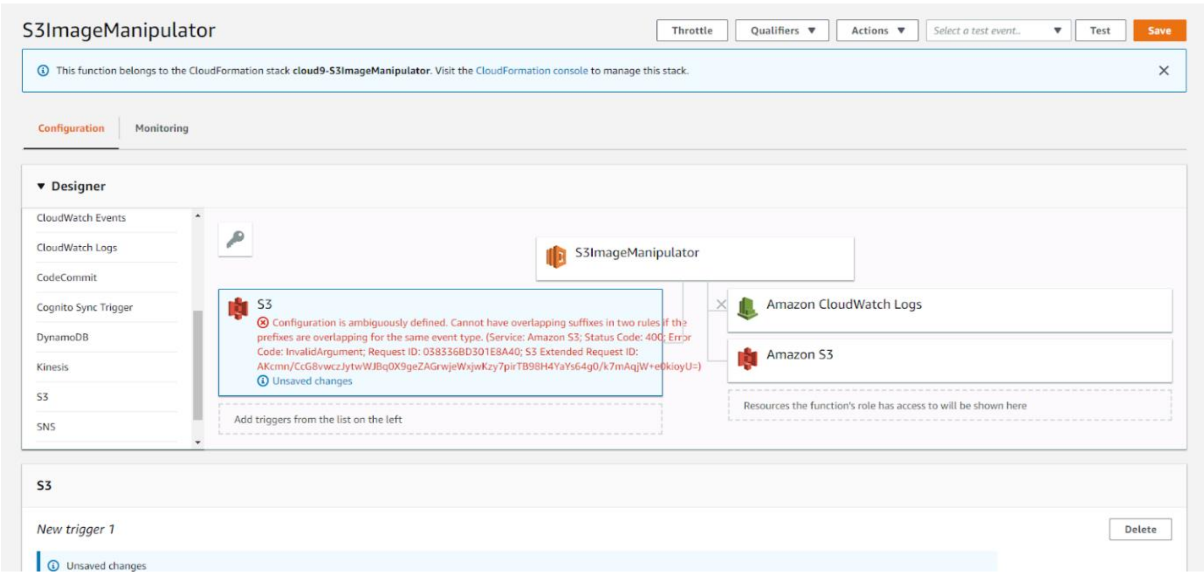
Now deploy the stack with the new definition from Cloud9 (this will be based on your modified template.yaml definition):



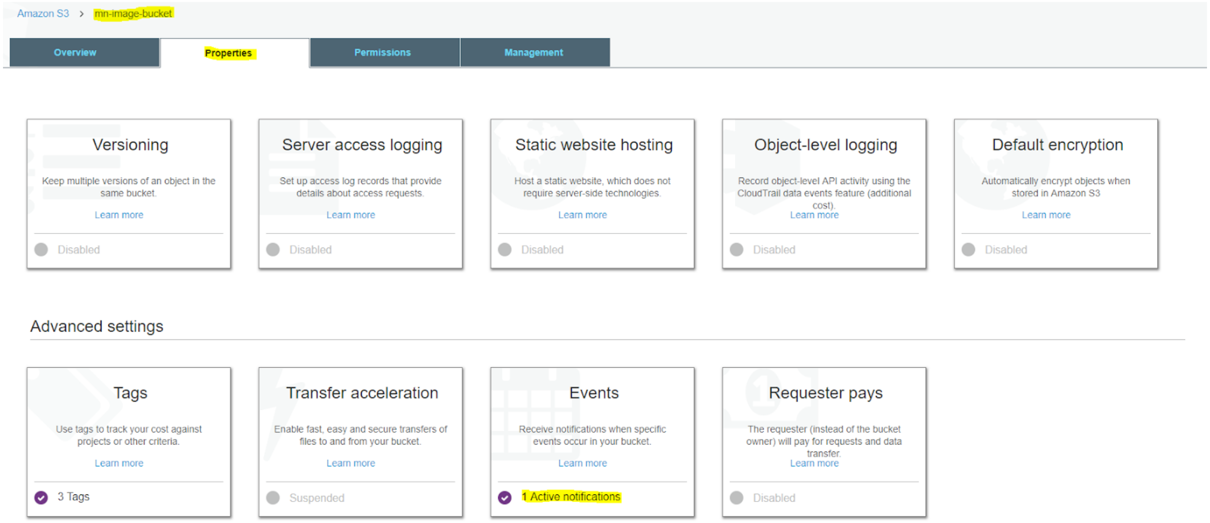
Note that after succesful deployment, the trigger will not show up in the Lambda console



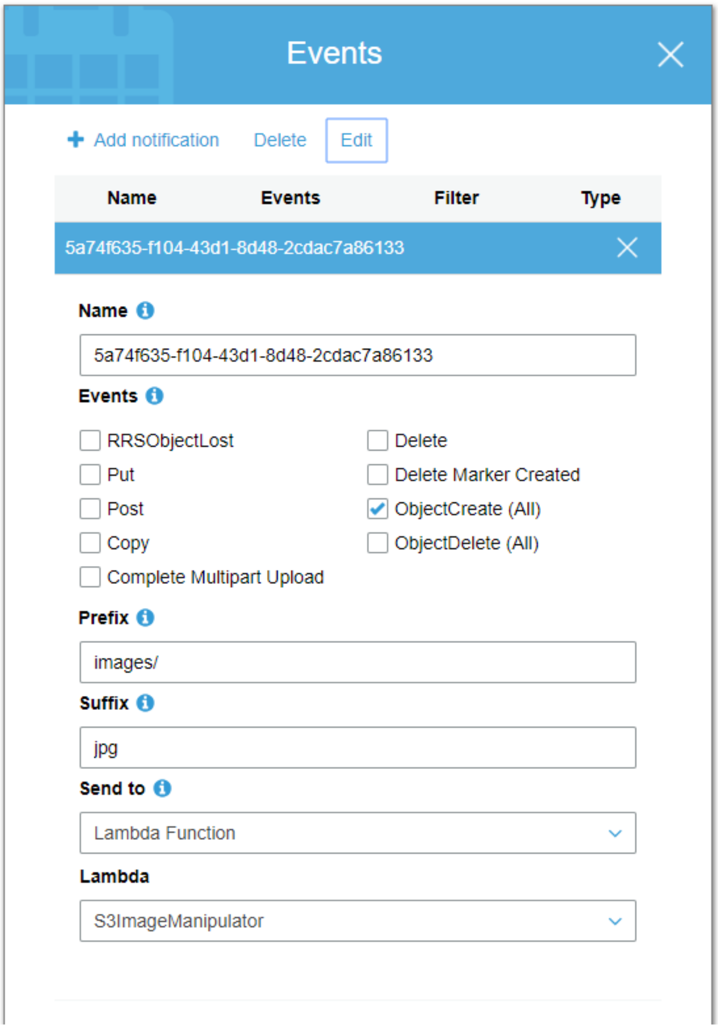
If you'd try to add the trigger manually again and saving the configuration, it makes lambda run into an error - stating the trigger is already defined:



Verify the properties for the input bucket on the S3 console for the defined events:

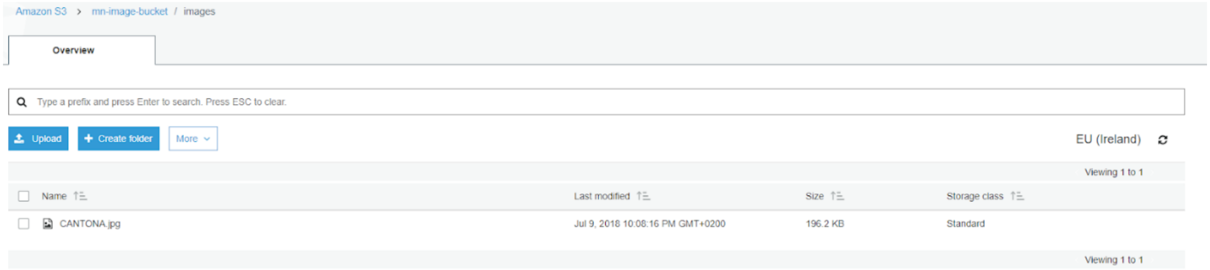


Inspecting the event shows that this is the event we have defined to trigger the lambda function:

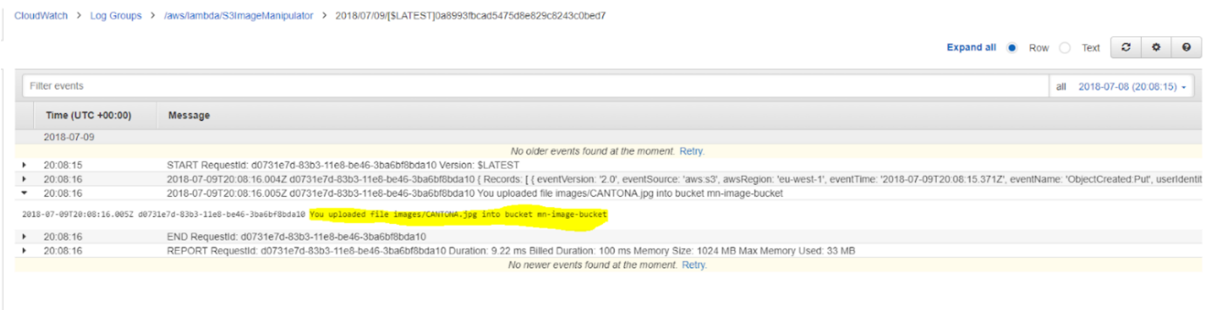


The proof of the pudding … is in the testing!

Let’s create a folder images in the source bucket and upload a file to it - see if it is really executed (assuming you have defined images as the prefix and jpg as the extension)



Wow!



Reading a file from S3

Now we need to actually retrieve the object designated in the S3 event into our program; the code to read an object from S3 is shown below – for now we don’t do any processing yet, but instead show the length of the object’s content. **Add the code to your lambda function from Cloud9 and redeploy your function.**

Code

```
const AWS = require('aws-sdk');
const util = require('util');

exports.handler = (event, context, callback) => {
    console.log(util.inspect(event, { showHidden: true, depth: null }));
    const S3 = new AWS.S3();
    const bucket = event.Records[0].s3.bucket.name;
    const filekey = event.Records[0].s3.object.key;

    var params = {
      Bucket: bucket,
      Key: filekey
     };

    S3.getObject(params, (err, data) => {
        if (err) {
            console.log(err);
            callback(err);
        } else {
            console.log("File size: " + data.ContentLength);
```

```
        };
    });
    callback();
};
```

Testdata structure

Adjust the content below to reflect your situation (must refer to existing file) to test GetObject call:

```
{
   "Records": [
      {
         "s3": {
            "bucket": {
               "name": "mn-image-bucket"
            },
            "object": {
               "key": "images/CANTONA.jpg"
            }
         }
      }
   ]
}
```

O, and TEST the code to see that it actually reads the file you worked so hard to upload ….

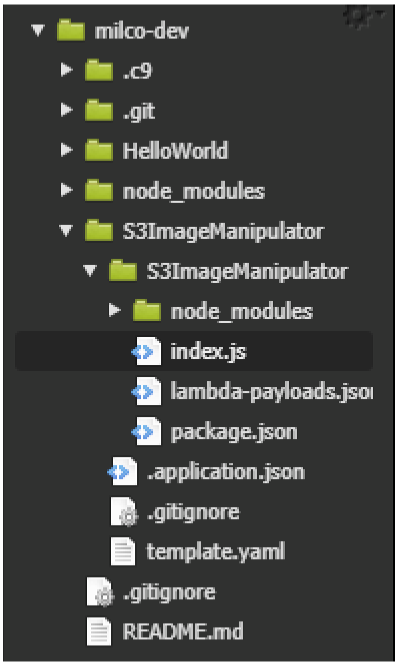
Next up:- image manipulation

To manipulate images, we will need to use external libraries from NPM, Jimp and async. In AWS Lambda, the functions are the unit of deployment, so we will need to install the libraries at the FUNCTION level: create the package.json file by performing an **npm init** from the function directory.

In my case "milco-dev" corresponded to /home/ec2user/environment, so I needed to:

open a terminal window from Cloud9
cd ~/environment/S3ImageManipulator/S3ImageManipulator
npm init
npm install jimp async --save

The result is that both modules are installed for the function and the dependencies are registered in the package.json file:



Jimp is needed for image manipulation, async is a utility library for invoking asynchronous functions in various ways; in this case, we will be invoking asynchronous functions sequentially after the previous one has completed.

According to the jimp documentation (https://www.npmjs.com/package/jimp), we can use the scan function on an image read by Jimp to manipulate an image on every pixel!

Restructure the code (in Cloud9):

```
const async = require('async');
const AWS = require('aws-sdk');
const jimp = require('jimp');
const S3 = new AWS.S3();

function downloadS3Object(S3Params, callback){
  console.time('S3GetObject');
  S3.getObject(S3Params, callback);
  console.timeEnd('S3GetObject');
}

function loadImageFromFile(S3Object, callback){
  console.time('loadImageFromFile');
  jimp.read(S3Object.Body, callback);
  console.timeEnd('loadImageFromFile');
}

function imageToBuffer(image, callback){
  console.time('imageToBuffer');
  image.getBuffer(jimp.AUTO, callback);
  console.timeEnd('imageToBuffer');
}
```

```
function uploadS3Object(params, image, next){
  console.time('uploadS3Object');
  // augment params for request
  params.Key = 'transformed/' + params.Key.split('/')[1];
  params.Body = image;
  params.ContentType = 'image/jpeg';  // easy access
  params.ACL = 'public-read';         // public access
  S3.putObject(params, next);
  console.timeEnd('uploadS3Object');
}

function handleError(err, next){
  if (err){
      console.log('Error:', err);
      throw(err);
  } else
    console.log('All OK');
}
/* Miscellaneous transformation functions, see Python examples at
 * https://t04glovern.github.io/2016/01/python-image-manipulation
 * - Image is represented as RGB+A, A=opacity/translucency
 * - Need to round the RGB values to integers, must be ≥0 and ≤ 255
 * Invert    : (R,G,B,A) -> (255-R,255-G,255-B,A)
 * Grayscale: (R,G,B,A) -> g = sum(R,G,B)/3; (g,g,g,A)
 * Sepia     : (R,G,B,A) -> v = 0.3 * R + 0.59 * G + 0.11 * B;  (2*v, 1.5*v, v, A)
 *
 *
 */
function invertImage(image, callback){
  console.time('invertImage');
  image.scan(0,0,image.bitmap.width, image.bitmap.height, (x,y,idx) => {
    // your implementation here ... see above for invert, sepia & grayscale
    // image.bitmap.data[ idx + 0 ] = ... ; // Red
    // image.bitmap.data[ idx + 1 ] = ... ; // Green
    // image.bitmap.data[ idx + 2 ] = ... ; // Blue
  }, callback);
  console.timeEnd('invertImage');
}

// this is the actual executable section
exports.handler = (event, context, callback) => {
    const srcImage = { Bucket: event.Records[0].s3.bucket.name,
                       Key: event.Records[0].s3.object.key
    };
    /*  Using async apply to inject an argument from the event
        into the async waterfall; both the download and upload
        functions need to know the bucket and the original
        image name.
        For the "drop-in-your-function-here" you could simply drop in an
        implemented function with the signature as in invertImage,
        e.g. function invertImage(image,callback).
     */
    async.waterfall(
       [ async.apply(downloadS3Object, srcImage)
       , loadImageFromFile
       , drop-in-your-function-here
       , imageToBuffer
       , async.apply(uploadS3Object, srcImage)
       ]
       , handleError
    );
};
```

Implement one (or more) function(s) for manipulating the image, even the InvertImage function is not completely implemented yet. Read the code to implement your image manipulation function of choice and drop your transformation function's name into the 'drop-in-your-function-here' placeholder. Deploy the stack to the cloud.

Test your function by uploading a new image into the upload bucket; verify that it has been transformed using the configuration function and inspect the CloudWatch logs.

If you want to drop in another function, you'll need to redeploy or make the change directly in the AWS Lambda console …

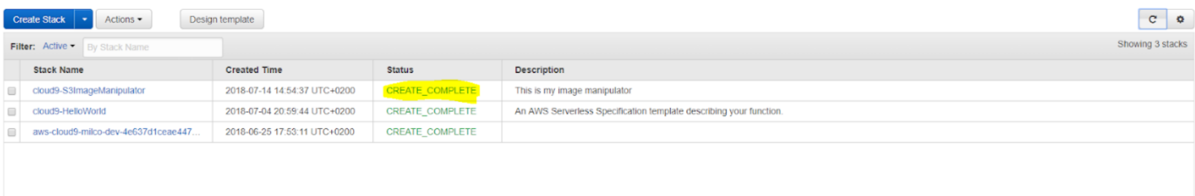
The proof of the pudding

- **delete the S3 buckets you've created from the S3 console**
- **delete the resources that CloudFormation created from the CloudFormation console by deleting the entire stack (we'll recreate this).**
- **Verify your IAM policy: have you specified the same S3 bucket name for the policy and in the CloudFormation template in your Lambda function definition?**
  (*this is where I went wrong upon testing my finished code*)

After you have succeeded, **deploy your Lambda function again from the Cloud9 IDE**; this will recreate all the required resources for running the lambda function (except for the IAM policy which we created by hand).

Adjust your IAM policy to allow the Lambda function to set an ACL on the object created, so you can directly view the image from S3

Check in the CloudFormation console that the stack provision has completed:



Next, create a new folder "images" inside your newly created bucket (or name it differently depending on the trigger prefix you specified in the template):

index.js | [λ] S3LambdaImageI | template.yaml

```yaml
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: This is my image manipulator
Resources:
  S3LambdaImageManipulator:
    Type: 'AWS::Serverless::Function'
    Properties:
      Handler: S3ImageManipulator/index.handler
      FunctionName: S3ImageManipulator
      Runtime: nodejs6.10
      CodeUri: .
      Description: This is my image manipulator
      MemorySize: 1024
      Timeout: 60
      Role: 'arn:aws:iam::612457436284:role/S3ImageManipulator'
      Events:
        ObjectCreatedEvent:
          Type: S3
          Properties:
            Bucket:
              Ref: MyInputBucket
            Events: s3:ObjectCreated:*
            Filter:
              S3Key:
                Rules:
                  - Name: prefix
                    Value: images/
                  - Name: suffix
                    Value: jpg
  MyInputBucket:
    Type: 'AWS::S3::Bucket'
    Properties:
      BucketName: milco-lambda-image-bucket
```

Before testing, revisit the IAM policy and add the "PutObjectACL" action to the policy; save the modified policy.

Policy ARN arn:aws:iam::612457436284:policy/S3MilcoLambdaImageBucketAccess

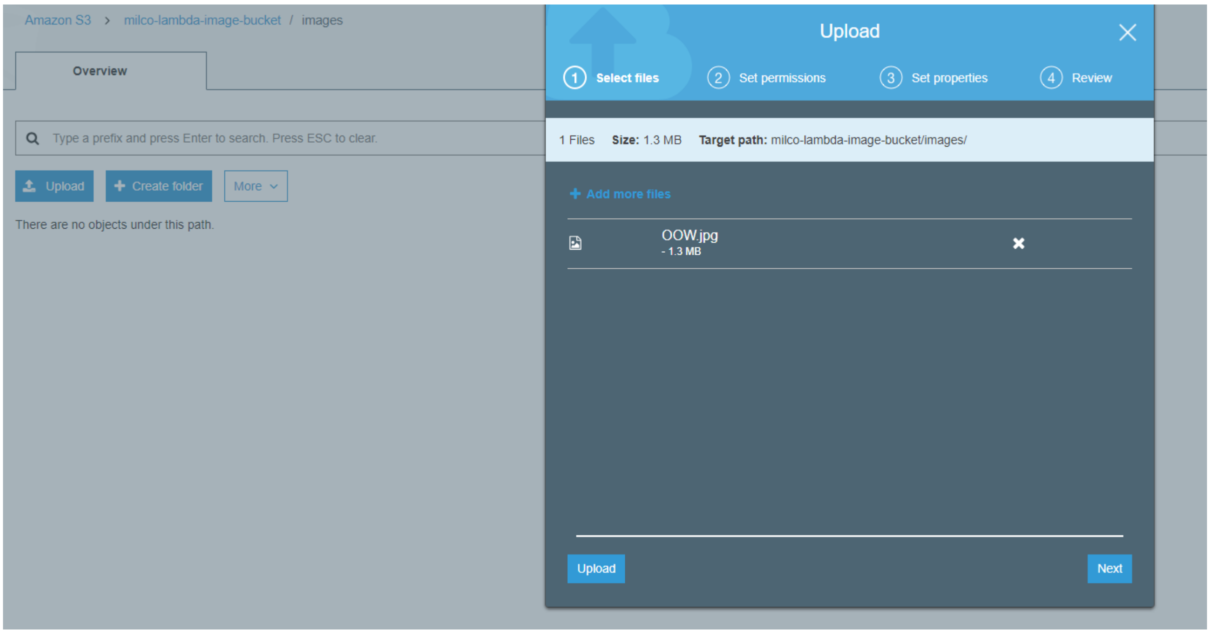
Description Allows access to my image bucket for reading and writing stuff

Permissions | Policy usage | Policy versions | Access Advisor

Policy summary | { } JSON | Edit policy

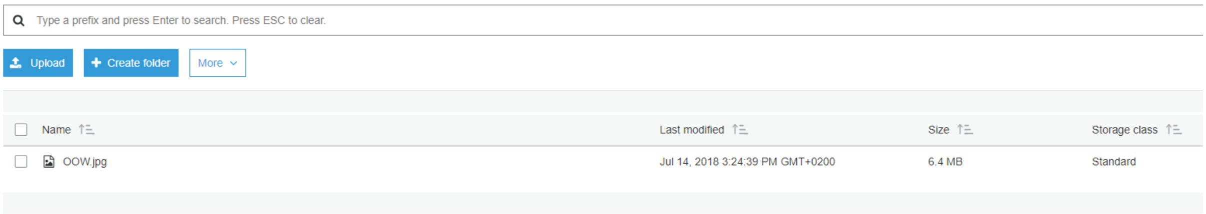
```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:GetObject",
                "s3:PutObjectAcl"
            ],
            "Resource": "arn:aws:s3:::milco-lambda-image-bucket/*"
        }
    ]
}
```

Now you're ready to test the lambda function: upload an image with the correct extension into your new folder.



My original image (1.3 MB)

The transformed image is quite larger than the original … probably should be adding a compression function as wel :-)



And the transformed (sepia) image:

Watching the Cloud

Checking the CloudWatch logs, it is shown that the function has been sized too large; max memory used is less than half of what is currently allocated. The function completes in slightly under 8 seconds, and is billed as 8000 ms@1GB:

```
START RequestId: 3db92666-8769-11e8-ae95-07db694f36f1 Version: $LATEST

2018-07-14T13:24:31.276Z 3db92666-8769-11e8-ae95-07db694f36f1 S3GetObject: 25.525ms

2018-07-14T13:24:38.822Z 3db92666-8769-11e8-ae95-07db694f36f1 uploadS3Object: 58.503ms

2018-07-14T13:24:38.822Z 3db92666-8769-11e8-ae95-07db694f36f1 imageToBuffer: 4759.048ms

2018-07-14T13:24:38.822Z 3db92666-8769-11e8-ae95-07db694f36f1 sepia: 5643.409ms

2018-07-14T13:24:38.822Z 3db92666-8769-11e8-ae95-07db694f36f1 loadImageFromFile: 7461.058ms

2018-07-14T13:24:39.186Z 3db92666-8769-11e8-ae95-07db694f36f1 All OK

END RequestId: 3db92666-8769-11e8-ae95-07db694f36f1

REPORT RequestId: 3db92666-8769-11e8-ae95-07db694f36f1 Duration: 7938.34 ms Billed Duration: 8000 ms Memory Size: 1024 MB Max Memory Used: 437 MB
```

Additional challenges

- Upload an image to the bucket and inspect the actual duration needed for the transformation (0-measurement).
  Adjust the memory allocated to your lambda function. Upload the original image again using another name - how does the memory allocation affect the actual duration?
- Implement another transformation function. Can you also configure dynamically which function to invoke, e.g. by using an environment variable on the lambda function's configuration holding the actual name of the transformation function to apply and retrieve the function object from the string, e.g. using eval or the global object (or find a more reliable way using Google) ?