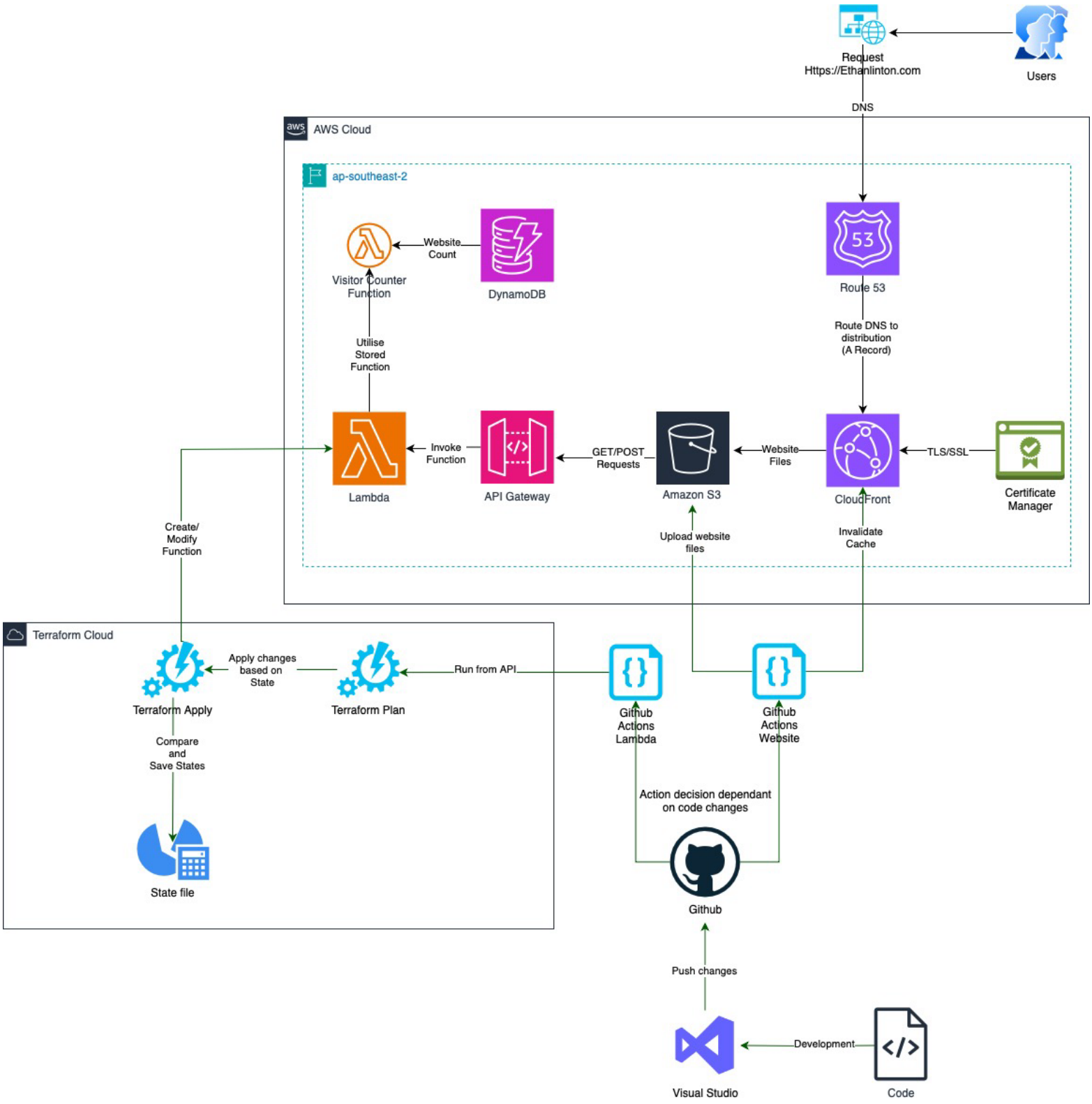


Amazon Web Services (AWS)
Multi-cloud
EthanLinton.com

Table of Contents

Introduction

Firstly, welcome to my portfolio website – ethanlinton.com. My goal of this website is to set up a platform where I can easily share my expertise and document projects I carry out in my free time.

This project – *Amazon Web Services (AWS) EthanLinton.com*, was set out to achieve the above goal. This document provides a brief overview of the entire tech stack that went into building my own portfolio website.

Tech stack

Please see the below list for all elements that went into the development of the overall architecture supporting and providing the website – ethanlinton.com.

Amazon Web Services

- Route 53
- CloudFront
- Certificate Manager
- Amazon S3
- API Gateway
- Lambda
- DynamoDB
- IAM
- IAM Identity Centre

GitHub

- Repository
- Actions
- Secret Variables

Terraform Cloud

- Organization
- Workspace
- State file

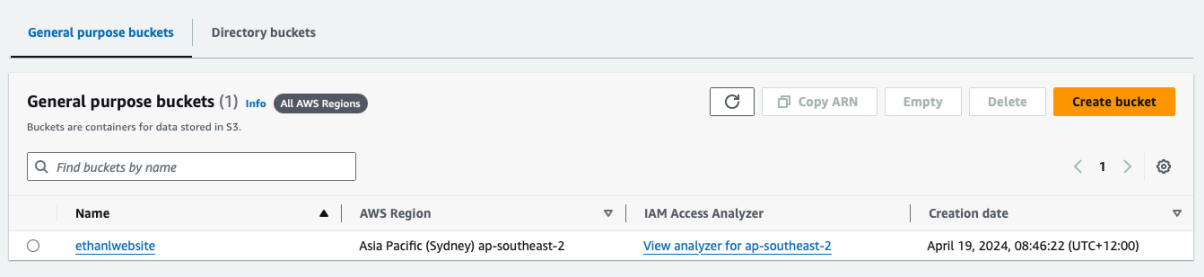
Service overview

This section aims to describe each service utilized in more detail. I will be covering my approach to implementation, and sharing screenshots and additional information as it relates to the specific feature providing benefits to the overall architecture and delivery of ethanlinton.com.

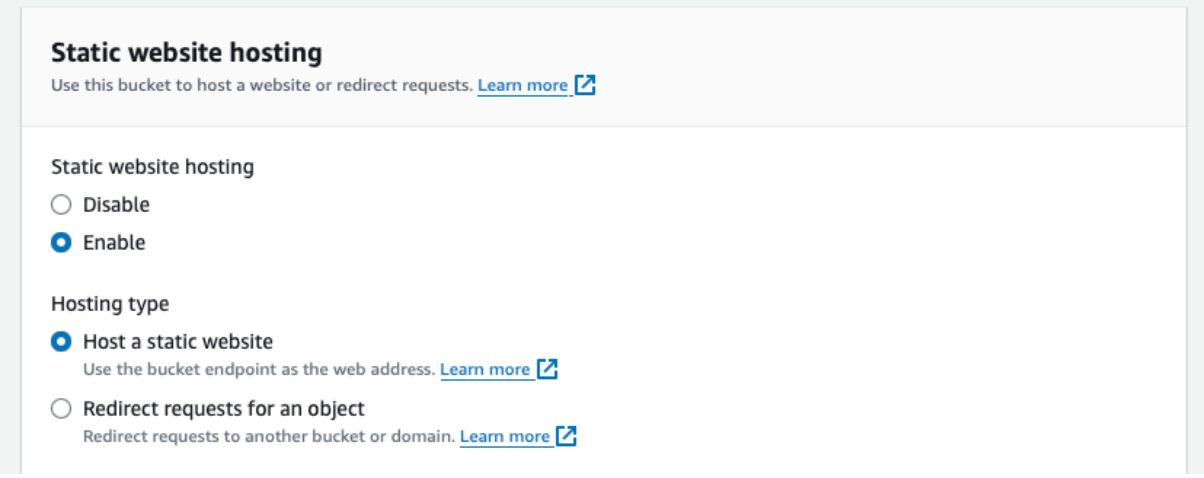
Amazon S3

Amazon S3 is my preferred choice of storing website files. I created a general purpose bucket in my chosen AWS region. Specifically, the S3 bucket stores the following files:

- HTML to serve the website
- CSS to customize the representation of data on the website
- Project files such as .pdfs
- Image files, including site assets, but also my resume.

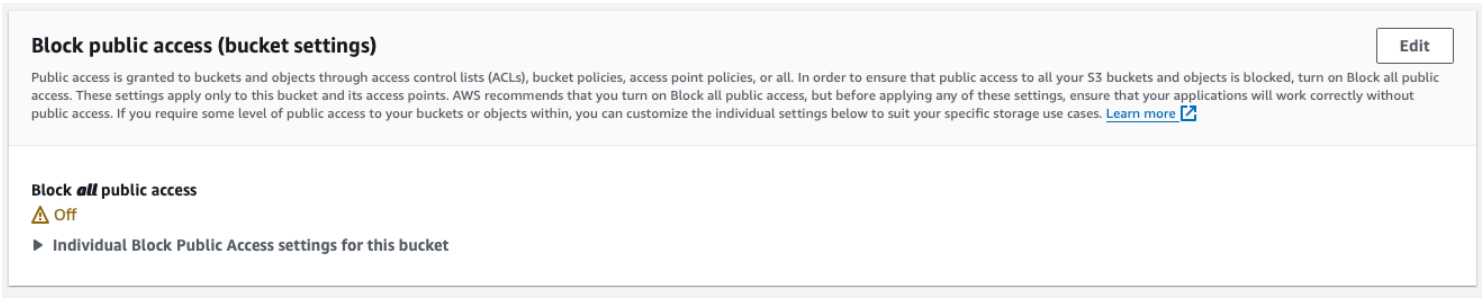


The S3 bucket is configured for static website hosting. Thus, the bucket is configured to use the bucket endpoint as the web address.



In order to host a static website, and serve website files via the Amazon S3 bucket, the bucket requires public access. Typical use cases of the S3 bucket require careful permission

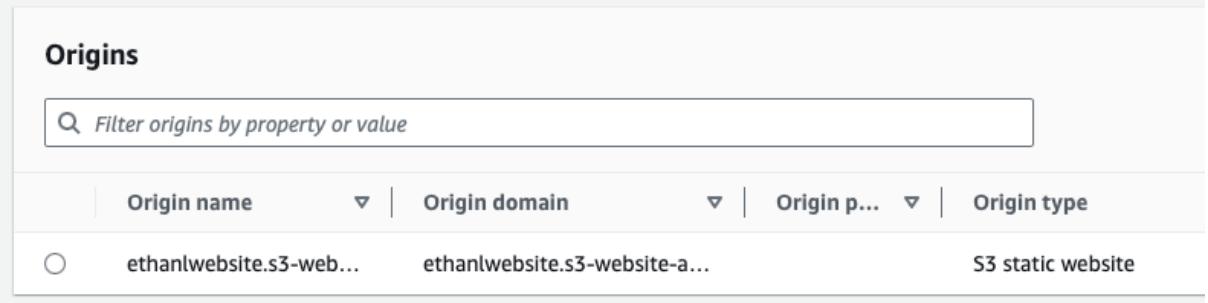
configuration, but in the case of hosting websites – disabling the below setting allows for the serving of website assets from the bucket to the public internet.



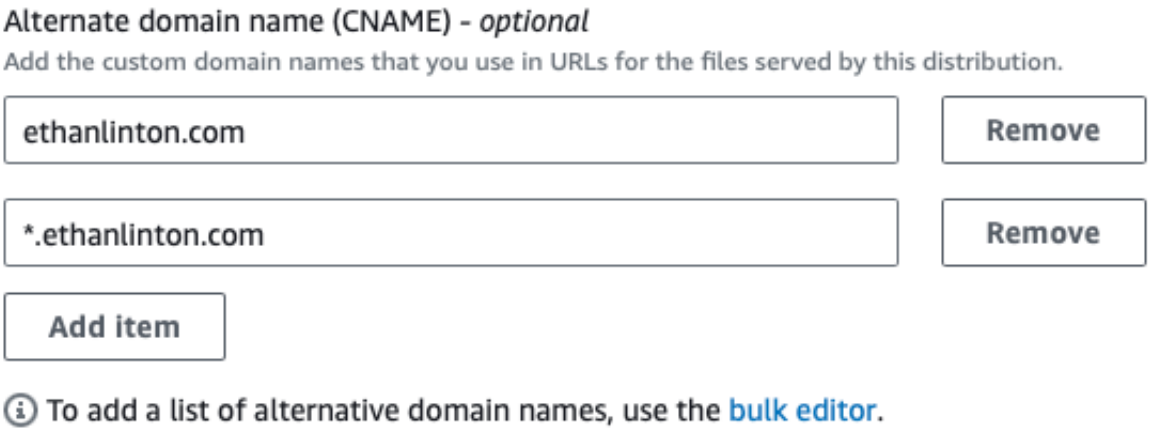
CloudFront

When it comes to content delivery networks – it only made sense to go with the Amazon CloudFront service due to the easy integration with S3 and Certificate manager.

During configuration of the CloudFront service – I utilized the S3 website endpoint as the Origin domain due to the requirement of static web hosting.

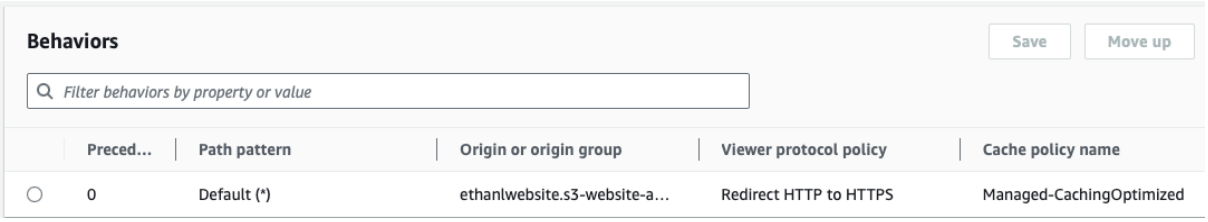


Next – I purchase my domain name via namecheap and supplied such domain name as an *alternative domain name (CNAME)* within my CloudFront distribution. Shown in the screenshot below – I added the root of the website as a domain name, but also added a wildcard of **.ethanlinton.com* to cover all possible patterns.



In order to enable HTTPS for the CloudFront distribution – I created a SSL certificate using AWS Certificate Manager. *Please see the AWS Certificate Manager section for more details.*

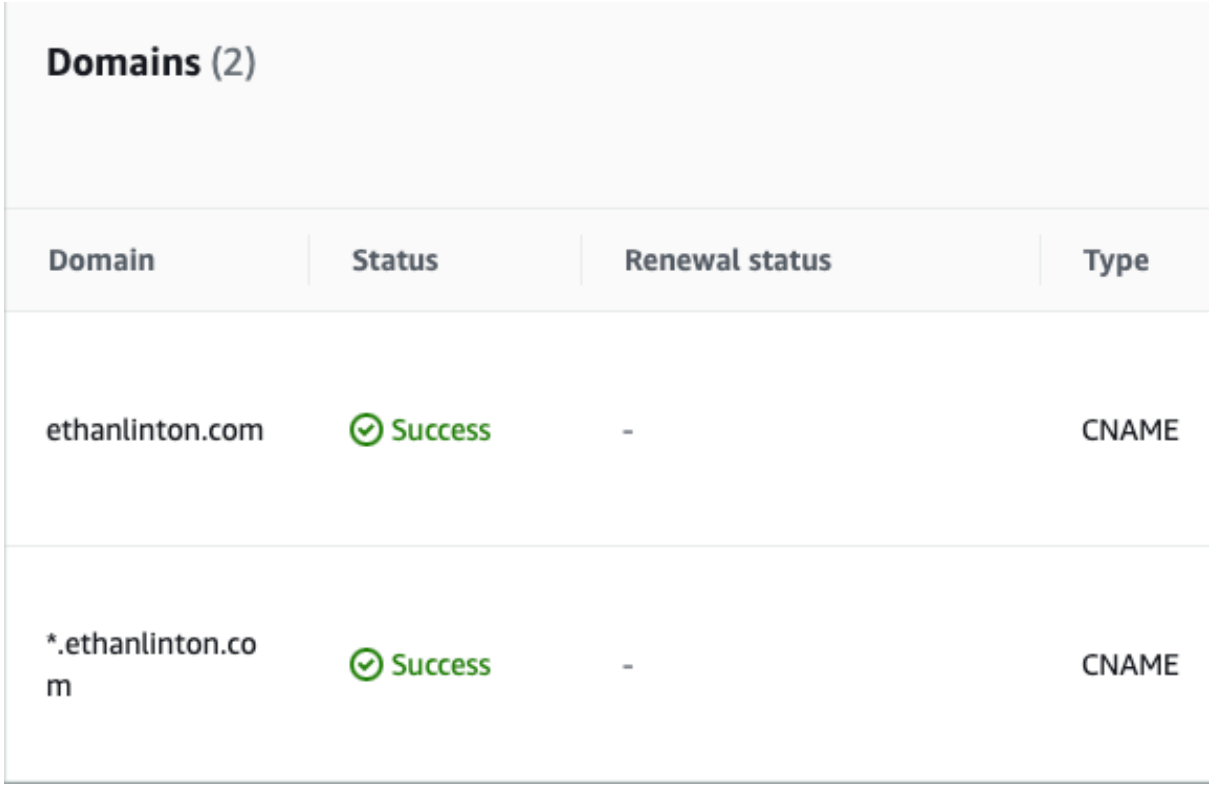
In order to force the protocol HTTPS, I added a CloudFront distribution behaviour to redirect HTTP to HTTPS for GET and HEAD methods.



Certificate Manager

I utilised AWS Certificate Manager to create a certificate in order to serve HTTPS to website visitors. I requested a public SSL/TLS certificate which since the certificate authority is Amazon – by default, the public certificate will be trusted by browsers and operating systems.

During creation of the public SSL/TLS certificate – I entered my fully qualified domain name being “ethanlinton.com”, but I also entered an additional domain in which includes a wild card – “*.ethanlinton.com”. Such use of wildcard ensures that future development surrounding the fully qualitied domain name such as the addition of “info.ethanlinton.com” will be SSL/TLS encrypted.



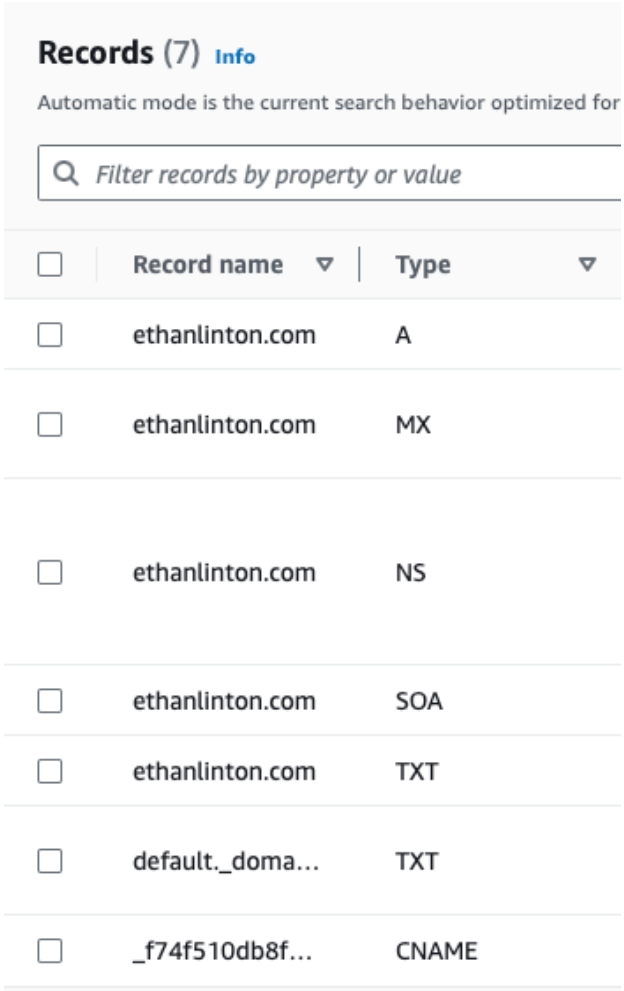
The validation method I used for validating domain ownership was DNS validation. Upon creation of the certificate – I inputted the appropriate DNS records (CNAME) which were created as part of the certificate into AWS Route 53. *Please see the Route 53 section for more details.*

Route 53

I originally purchased my domain via Namecheap – so in order to utilise Route 53 with my domain, I was required to add my 4 custom nameservers (NS) into my Namecheap dashboard. Upon entering the nameservers, I waited roughly 30 minutes for DNS propagation to take effect.

The Route 53 hosted zone for "Ethanlinton.com" contains a total for 7 records. The below list explains each record:

1. A Record – routes traffic to AWS CloudFront instance
2. MX Record – routes traffic to my custom email account hello@ethanlinton.com
3. NS Record – routes traffic to the correct DNS servers
4. SOA Record – designates the primary NS value
5. TXT Record – custom email record, sender policy framework
6. TXT Record – custom email record, domain generated key
7. CNAME Record – inputted record from SSL/TLS certificate generated by AWS

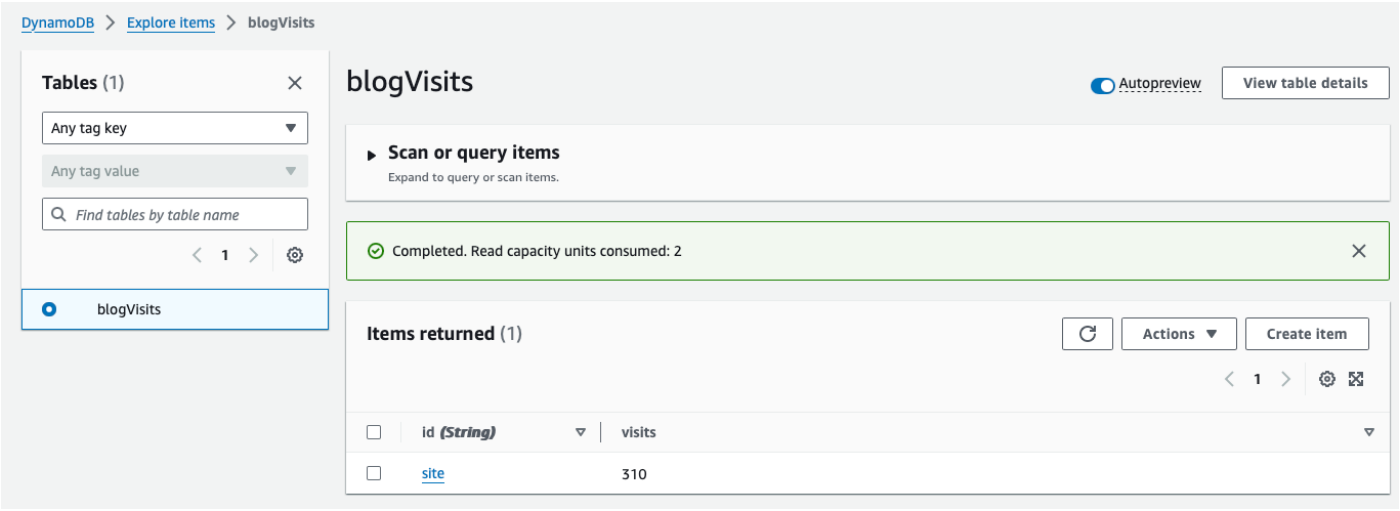


DynamoDB

Surrounding the implementation of the website visitor counter – I chose AWS DynamoDB as the backend to store website visitor counter data.

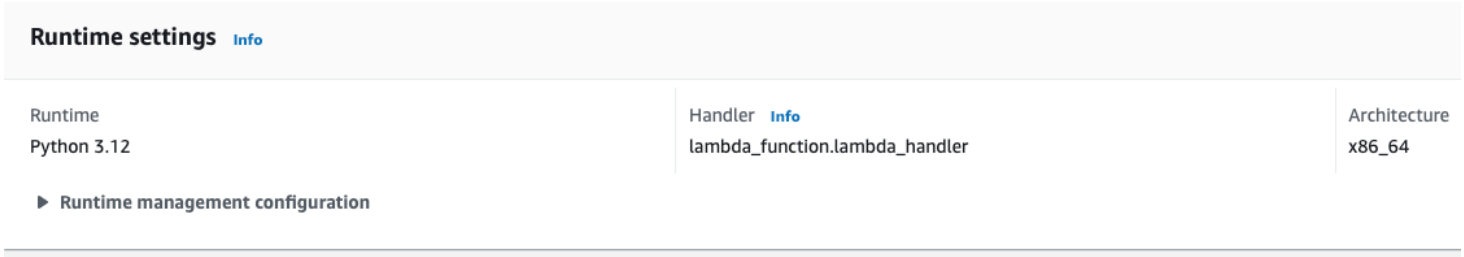
During creation of the DynamoDB table – I named my table “blogVisits” and set my partition key as string variable “id”. During configuration of the “blogVisits” table – I created a primary key called “site”. I added another attribute to the table called “visits” and stored it as a number variable. The visits attribute will hold all relevant website visitor counter data – in this case a simple counter that goes up per visit to “Ethanlinton.com”.

The below screenshot shows the contents of the table items within the blogVisits table stored within AWS DynamoDB.

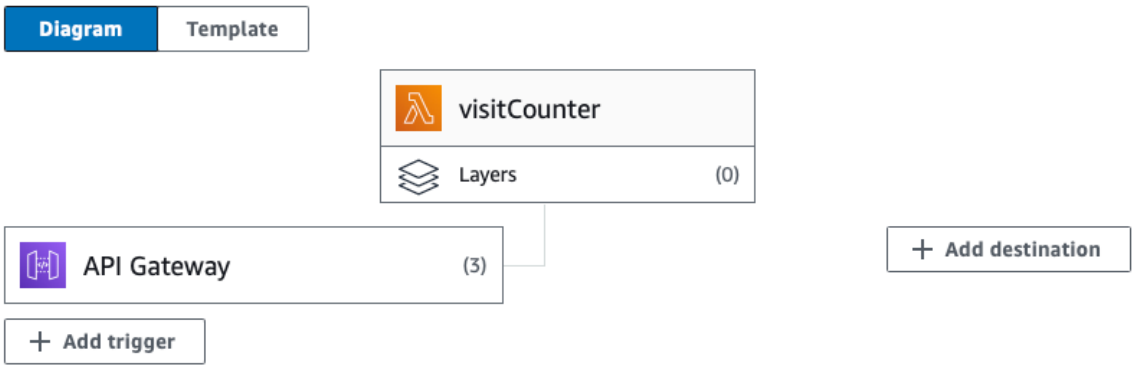


Lambda

I utilised AWS Lambda to execute my visitor counter code in response to site visitors. During creation of the Lambda Function, since I will be writing my visitor counter code in Python, my runtime for such function is set to “Python 3.12”.



The Lambda function is triggered via the AWS API Gateway service. *Please see API Gateway section for more details on API Gateway configuration*.



The below code is the code used within the Lambda function. This script interacts with an AWS DynamoDB table to keep track of the number of visits to Ethanlinton.com. Here's a simple breakdown of what it does:

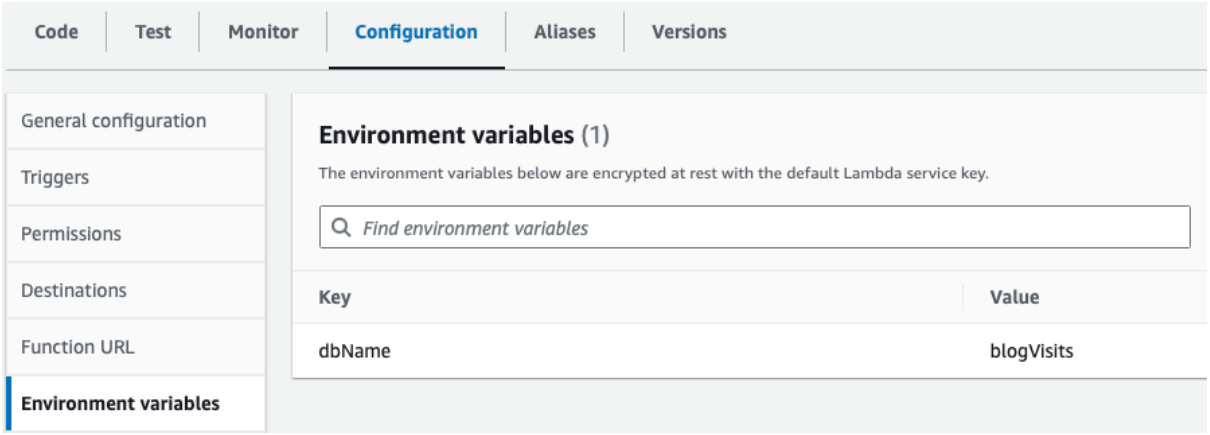
1. get_ddb_table(): Retrieves the DynamoDB table specified in the environment variable "dbName".
2. get_count(table): Queries the DynamoDB table to get the current number of visits recorded for the website.
3. lambda_handler(event, context): This function is the entry point for an AWS Lambda function. It updates the visit count in the DynamoDB table by incrementing it and returns an HTTP response containing the updated visit count.

It increments the "visits" attribute for the "site" item in the DynamoDB table. Constructs and returns an HTTP response with status code 200, allowing cross-origin requests, and returning the updated visit count in the response body.

See below for the full code:

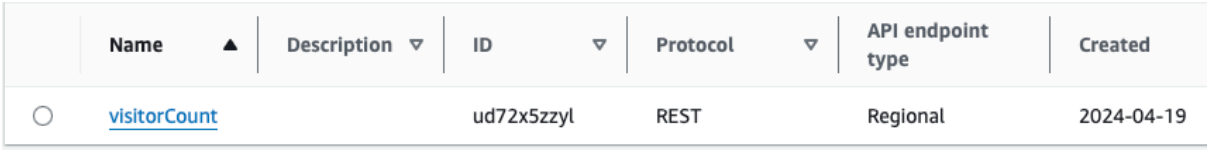
```python
import boto3
import os
import json
from boto3.dynamodb.conditions import Key

def get_ddb_table():
    # Get the table name from environment variables
    databaseName = os.environ.get("dbName")
    # Create a DynamoDB resource
    dynamodb = boto3.resource("dynamodb")
    # Retrieve the specified DynamoDB table
    table = dynamodb.Table(databaseName)
    return table

# Returns the latest value of the 'visits' attribute for the 'site' item
def get_count(table):
    # Query the DynamoDB table to get the 'visits' value for the 'site' item
    response = table.query(KeyConditionExpression=Key("id").eq("site"))
    # Extract the 'visits' value from the response
    count = response["Items"][0]["visits"]
    return count

# Increment the 'visits' value for the 'site' item
def lambda_handler(event, context):
    # Get the DynamoDB table
    table = get_ddb_table()
    # Update the 'visits' value for the 'site' item by incrementing it
    response = table.update_item(
        Key={"id": "site"},
        UpdateExpression="ADD visits :incr",
        ExpressionAttributeValues={":incr": 1},
        ReturnValues="UPDATED_NEW"
    )

    # Return the HTTP response
    return {
        "statusCode": 200,
        "headers": {
            "Access-Control-Allow-Origin": "*",
            "Access-Control-Allow-Headers": "Content-Type,X-Amz-Date,Authorization,X-Api-Key,X-Amz-Security-Token",
            "Access-Control-Allow-Credentials": "true",
            "Content-Type": "application/json"
        },
        'body': get_count(table)  # Return the latest 'visits' value in the response body
    }
```

Noting the *databaseName = os.envion.get("dbName")* line – I set an environmental variable under function configuration that points to the correct DynamoDB table name.

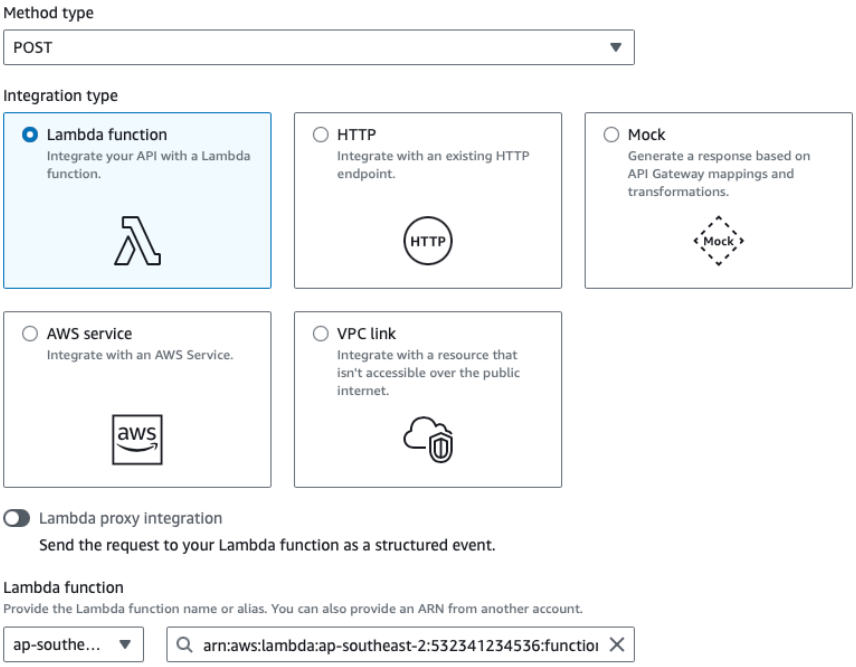


API Gateway

In order to invoke the lambda function I configured a REST API via the AWS API Gateway service to initiate the function upon request from website, and respond with the latest website visitor count data.



During configuration of the newly created REST API – I created a new POST method, selecting the Lambda function integration type and chose the "visitCounter" lambda function I created earlier.



Upon creating the POST method within the REST API – I tested the method against my selected lambda function within such method being “visitCounter”.

/ - POST method test results

Request	Latency ms	Status
/	2928	200

Response body


```
{"statusCode": 200, "headers": {"Access-Control-Allow-Origin": "*", "Access-Control-Allow-Headers":
"Content-Type,X-Amz-Date,Authorization,X-Api-Key,X-Amz-Security-Token", "Access-Control-Allow-
Credentials": "true", "Content-Type": "application/json"}, "body": 337}
```


After successfully passing POST method testing – I deployed the API and copied the invoke URI into https://reqbin.com which is an online browser REST API test tool. Below are the results from testing the invoke URI.

Status: 200 (OK) Time: 1279 ms Size: 0.26 kb

Content (10) Headers (9) Raw (11) JSON Timings


```
{
    "statusCode": 200,
    "headers": {
        "Access-Control-Allow-Origin": "*",
        "Access-Control-Allow-Headers": "Content-Type,X-Amz-Date,Authorization,X-Api-Key,X-Amz-Security-Token",
        "Access-Control-Allow-Credentials": "true",
        "Content-Type": "application/json"
    },
    "body": 339
}
```


Before adding the required script into my HTML code for EthanLinton.com – I enabled Cross-Origin Resource Sharing (CORS) support within my REST API resource. Adding such support to my REST API ensures that requests that are initiated from scripts running the a browser are blocked. During configuration of CORs – I allowed the OPTIONS and GET methods.

Methods (3) Delete Create method

Method type	Integration type	Authorization	API key
GET	Lambda	None	Not required
OPTIONS	Mock	None	Not required
POST	Lambda	None	Not required

I next added the HTML code required to reach out to my REST API in order to invoke the lambda function and get a visit counter response. I added the below apiURL which is used to interact with the REST API.

```
<script type = "text/javascript">
                    var apiUrl = "https://ud72x5zzyl.execute-api.ap-southeast-
2.amazonaws.com/counters/";
```

CI/CD Website flow

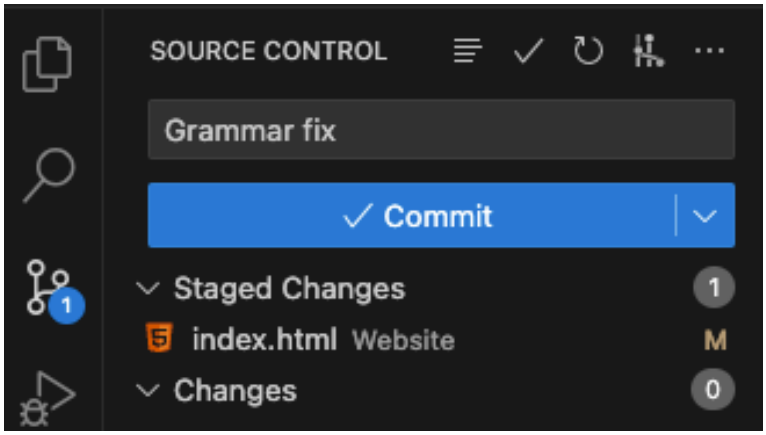
This section describes the CI/CD flow I created to assist with development of EthanLinton.com.

My development workflow surrounding website assets includes the following:

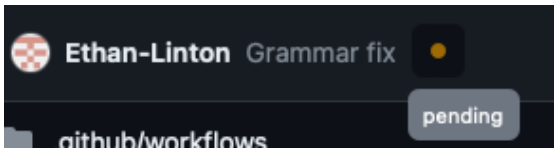
- Visual studio code
- Github.

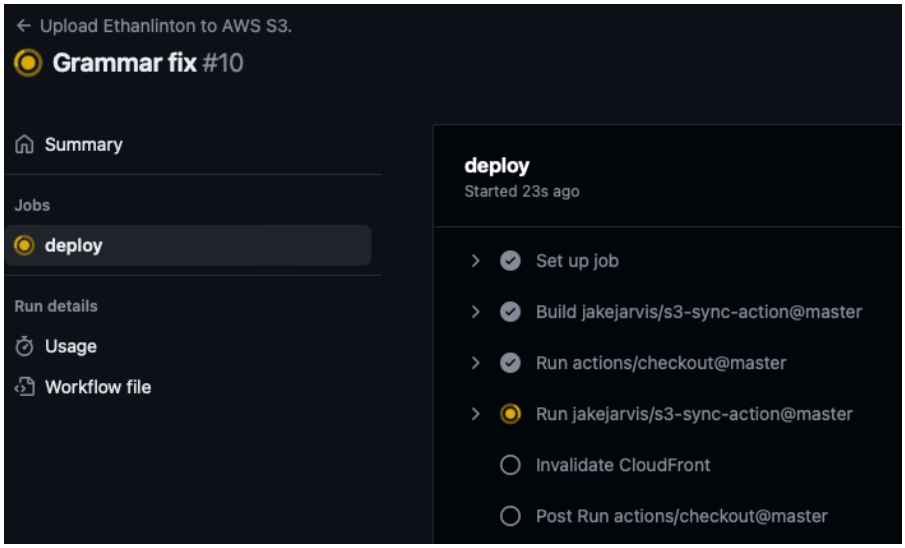
The overall process of the workflow is the following:

1. Make modifications to website code via Visual studio code
2. Push changes to GitHub repository

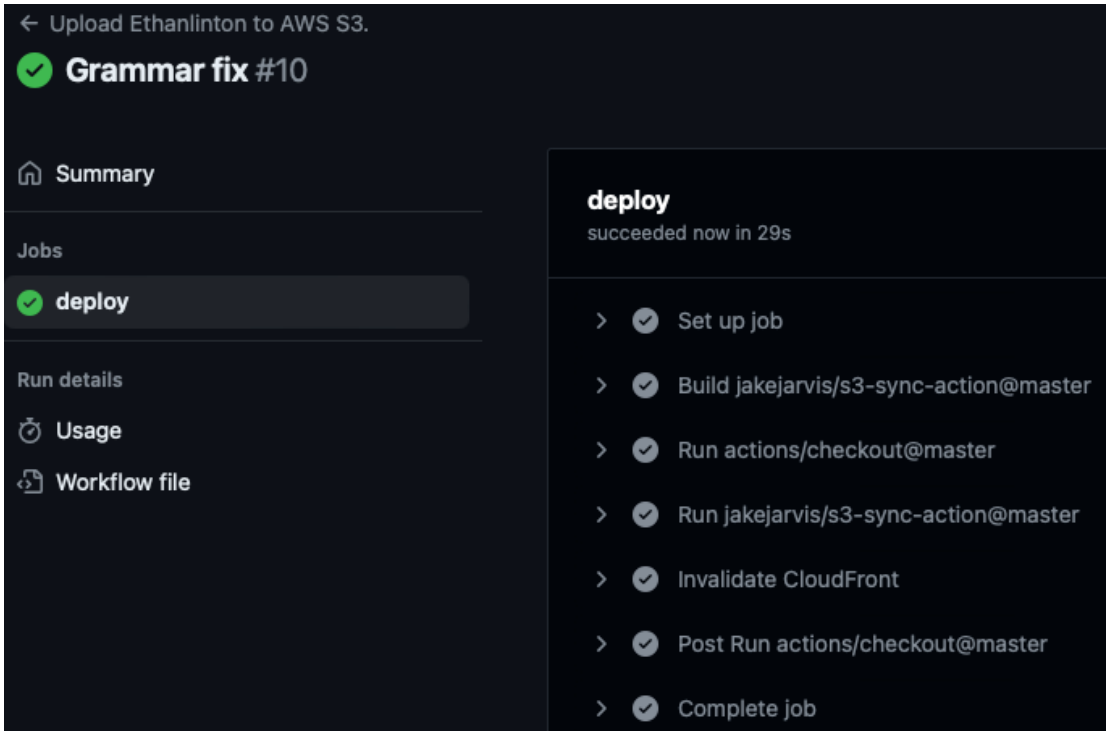


3. The GitHub repository will initiate GitHub actions





4. GitHub actions will automatically upload the modified files to S3, and invalidate the CloudFront cache to refresh the website.



As you can see from the above topology diagram on Page 1, I make use of two GitHub action workflows. The choice of which workflow to automatically start is dependant on what I push to GitHub. For example, the website CI/CD workflow .yml file contains the following snippet of code:

```
on:
  push:
    paths:
      - Website/**
```

This ensures that such workflow is only run if changes are made within my Website/ directory.

The below code shows the entirety of my workflow for deploying changes automatically to my website:

```
name: Upload Ethanlinton to AWS S3.

on:
  push:
    paths:
      - Website/**

jobs:
  deploy:
    runs-on: ubuntu-latest
```

```yaml
steps:
- uses: actions/checkout@master
- uses: jakejarvis/s3-sync-action@master
  with:
    args: --acl private --follow-symlinks --delete
  env:
    AWS_S3_BUCKET: ${{ secrets.AWS_S3_BUCKET }}
    AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
    AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
    AWS_REGION: 'ap-southeast-2'
    SOURCE_DIR: 'Website'

- name: Invalidate CloudFront
  uses: chetan/invalidate-cloudfront-action@v2
  env:
    DISTRIBUTION: ${{ secrets.DISTRIBUTION }}
    PATHS: "/*"
    AWS_REGION: "ap-southeast-2"
    AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
    AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
```

You'll notice the usage of environmental variables surrounding data relating to authentication, and authorisation. Such environmental variables are stored as secrets within my GitHub repository. Each time the workflow is run, it will read the secrets stored within my repository settings and apply them to the workflow – thus allowing access to my AWS instances.

CI/CD Lambda function flow

This section describes the CI/CD flow I created to assist with development of lambda functions for Ethanlinton.com.

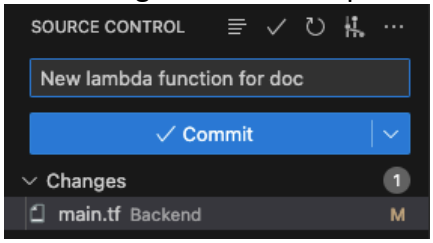
My development workflow surrounding website assets includes the following:

- Visual studio code
- Github
- Terraform cloud

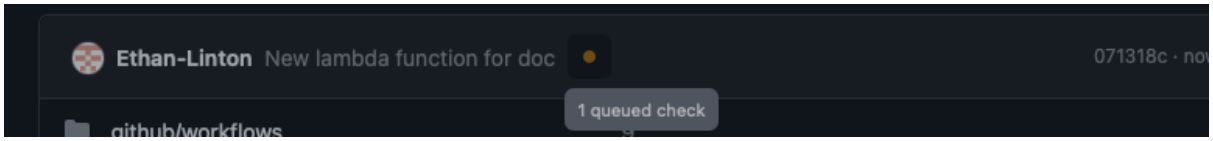
The choice of utilising Terraform cloud instead of AWS SAM, was on interest surrounding the specific technology and also enabling this project to be multi-cloud based.

The overall process of the workflow is the following:

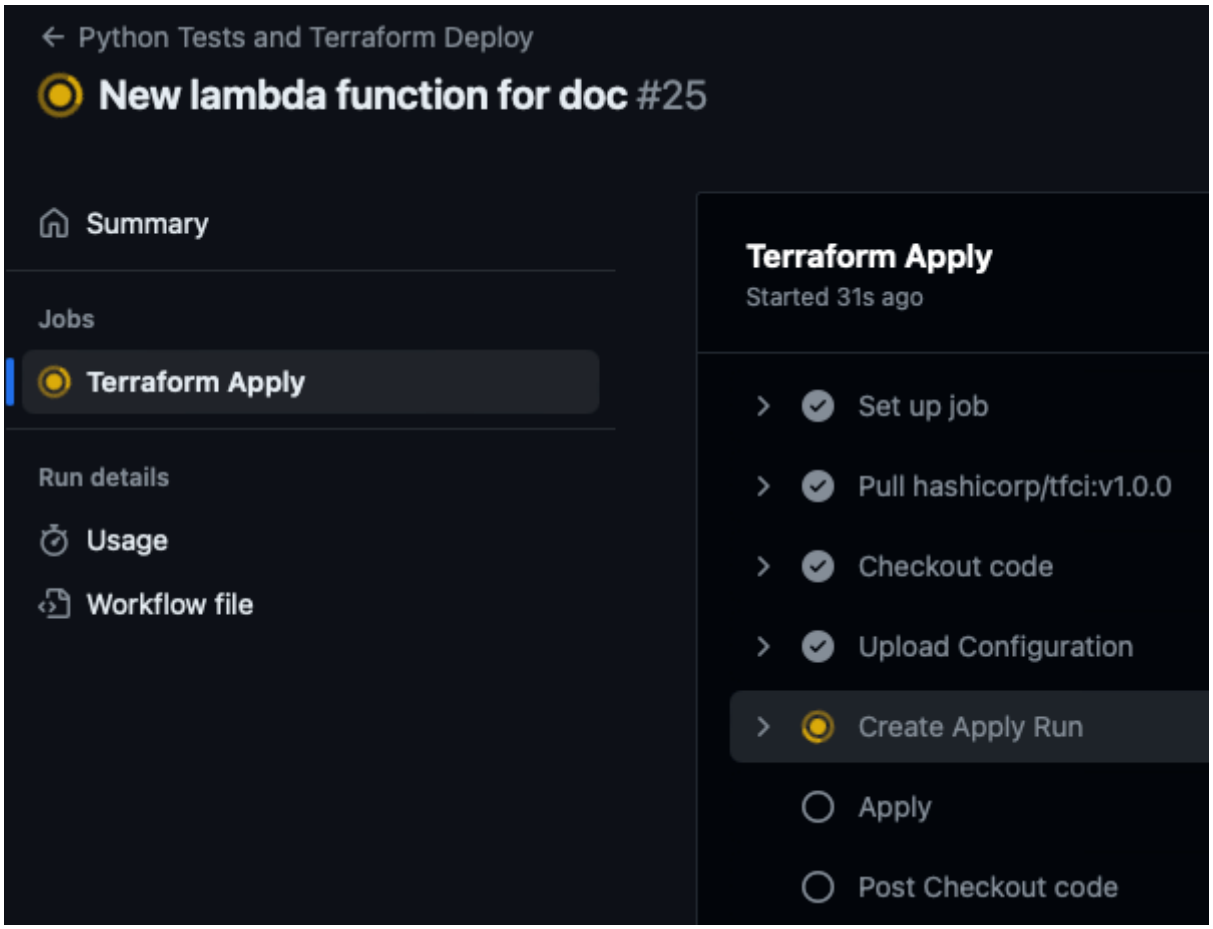
1. Modify backend lambda code/create new lambda function
2. Push changes to GitHub repository

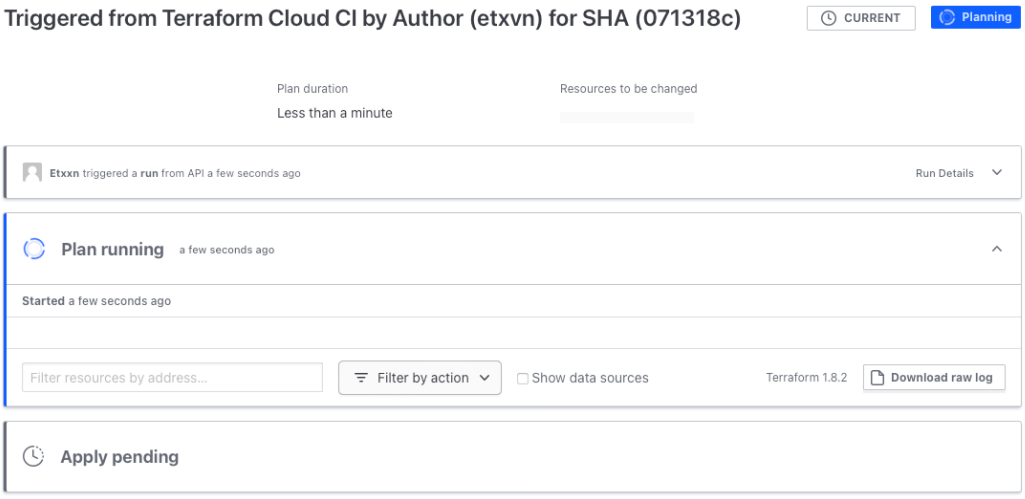


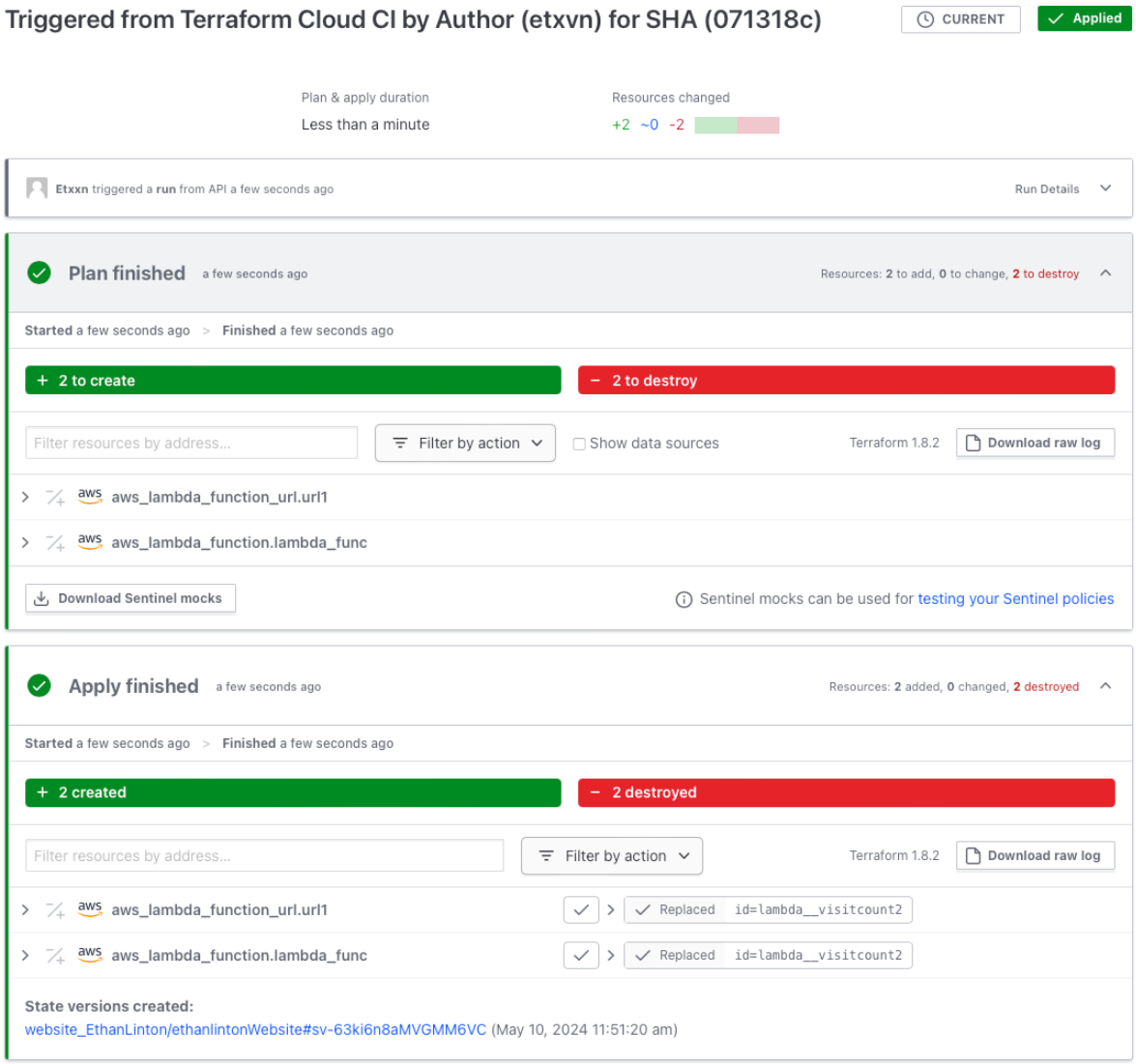
3. GitHub repository will initiate GitHub Actions

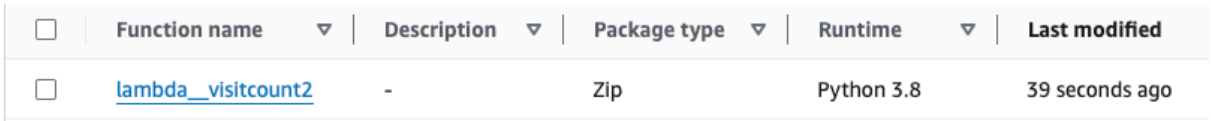


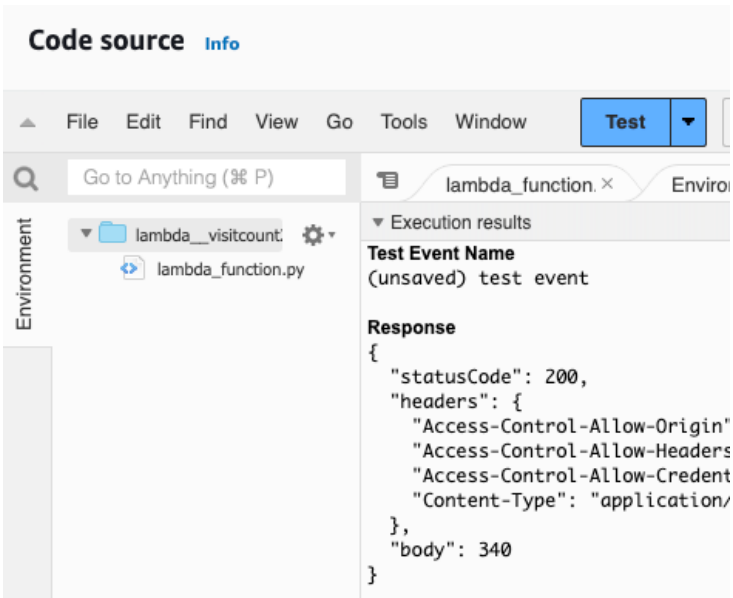
4. GitHub actions will invoke Terraform cloud to begin planning for the change, automatically apply the changes needed dependant on state file, and push the written modifications to AWS Lambda.






Triggered from Terraform Cloud CI by Author (etxvn) for SHA (071318c)
CURRENT
Applied
Plan & apply duration
Less than a minute
Resources changed
+2 ~0 -2
Etxxn triggered a run from API a few seconds ago
Run Details
Plan finished a few seconds ago
Resources: 2 to add, 0 to change, 2 to destroy
Started a few seconds ago > Finished a few seconds ago
+ 2 to create
− 2 to destroy
Filter resources by address...
Filter by action
Show data sources
Terraform 1.8.2
Download raw log
aws_lambda_function_url.url1
aws_lambda_function.lambda_func
Download Sentinel mocks
Sentinel mocks can be used for testing your Sentinel policies
Apply finished a few seconds ago
Resources: 2 added, 0 changed, 2 destroyed
Started a few seconds ago > Finished a few seconds ago
+ 2 created
− 2 destroyed
Filter resources by address...
Filter by action
Terraform 1.8.2
Download raw log
aws_lambda_function_url.url1
Replaced id=lambda__visitcount2
aws_lambda_function.lambda_func
Replaced id=lambda__visitcount2
State versions created:
website_EthanLinton/ethanlintonWebsite#sv-63ki6n8aMVGMM6VC (May 10, 2024 11:51:20 am)


Function name
Description
Package type
Runtime
Last modified
lambda__visitcount2
-
Zip
Python 3.8
39 seconds ago


Code source Info
File Edit Find View Go Tools Window
Test
Go to Anything (⌘ P)
lambda_function
Environment
lambda__visitcount
lambda_function.py
Execution results
Test Event Name
(unsaved) test event
Response
{
"statusCode": 200,
"headers": {
"Access-Control-Allow-Origin"
"Access-Control-Allow-Headers
"Access-Control-Allow-Credent
"Content-Type": "application/
},
"body": 340
}

The same concept applied to the website CI/CD workflow was applied to the Lambda function CI/CD workflow. Workflow initiation was dependant on what modifications were made. In order to initiate the lambda function CI/CD workflow I must make changes within the backend folder.

```
on:
    push:
      paths:
        - Backend/**
```

See below for the full CI/CD workflow .yml surrounding Lambda function deployment:

```
name: Python Tests and Terraform Deploy

on:
    push:
      paths:
        - Backend/**
env:
  TF_CLOUD_ORGANIZATION: "website_EthanLinton"
  TF_API_TOKEN: "${{ secrets.TF_API_TOKEN }}"
  TF_WORKSPACE: "ethanlintonWebsite"
  CONFIG_DIRECTORY: "./Backend"

jobs:
  terraform:
    if: github.repository != 'hashicorp-education/learn-terraform-github-actions'
    name: "Terraform Apply"
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: ./Backend
    permissions:
      contents: read
    steps:
    - name: Checkout code
      uses: actions/checkout@v2
    - name: Upload Configuration
      uses: hashicorp/tfc-workflows-github/actions/upload-configuration@v1.0.0
      id: apply-upload
      with:
        workspace: ${{ env.TF_WORKSPACE }}
        directory: ${{ env.CONFIG_DIRECTORY }}
    - name: Create Apply Run
      uses: hashicorp/tfc-workflows-github/actions/create-run@v1.0.0
      id: apply-run
      with:
        workspace: ${{ env.TF_WORKSPACE }}
        configuration_version: ${{ steps.apply-
upload.outputs.configuration_version_id }}
    - name: Apply
      uses: hashicorp/tfc-workflows-github/actions/apply-run@v1.0.0
```

```
      if: fromJSON(steps.apply-
run.outputs.payload).data.attributes.actions.IsConfirmable
      id: apply
      with:
        run: ${{ steps.apply-run.outputs.run_id }}
        comment: "Apply Run from GitHub Actions CI ${{ github.sha }}"
```

Environmental variables are stored as secrets within the GitHub repository for both AWS and Terraform credentials.

Future work

This project will always be on-going – EthanLinton.com will continue to evolve as it is the hub for all my other projects.

The future work I have in mind surrounding the on-going development of EthanLinton.com is as follows:

- Automated testing for both CI/CD workflows
    - Test lambda function deployment via GitHub using Moto to simulate AWS infrastructure
    - Code check website code to ensure format is correct
- Implement a content management system (CMS) into EthanLinton.com
    - Currently I document all projects via Microsoft Word and upload via PDF
    - I want to make this workflow more efficient via adding CMS to enable quicker documentation.