SERVERLESS API PROXY IN THE CLOUD

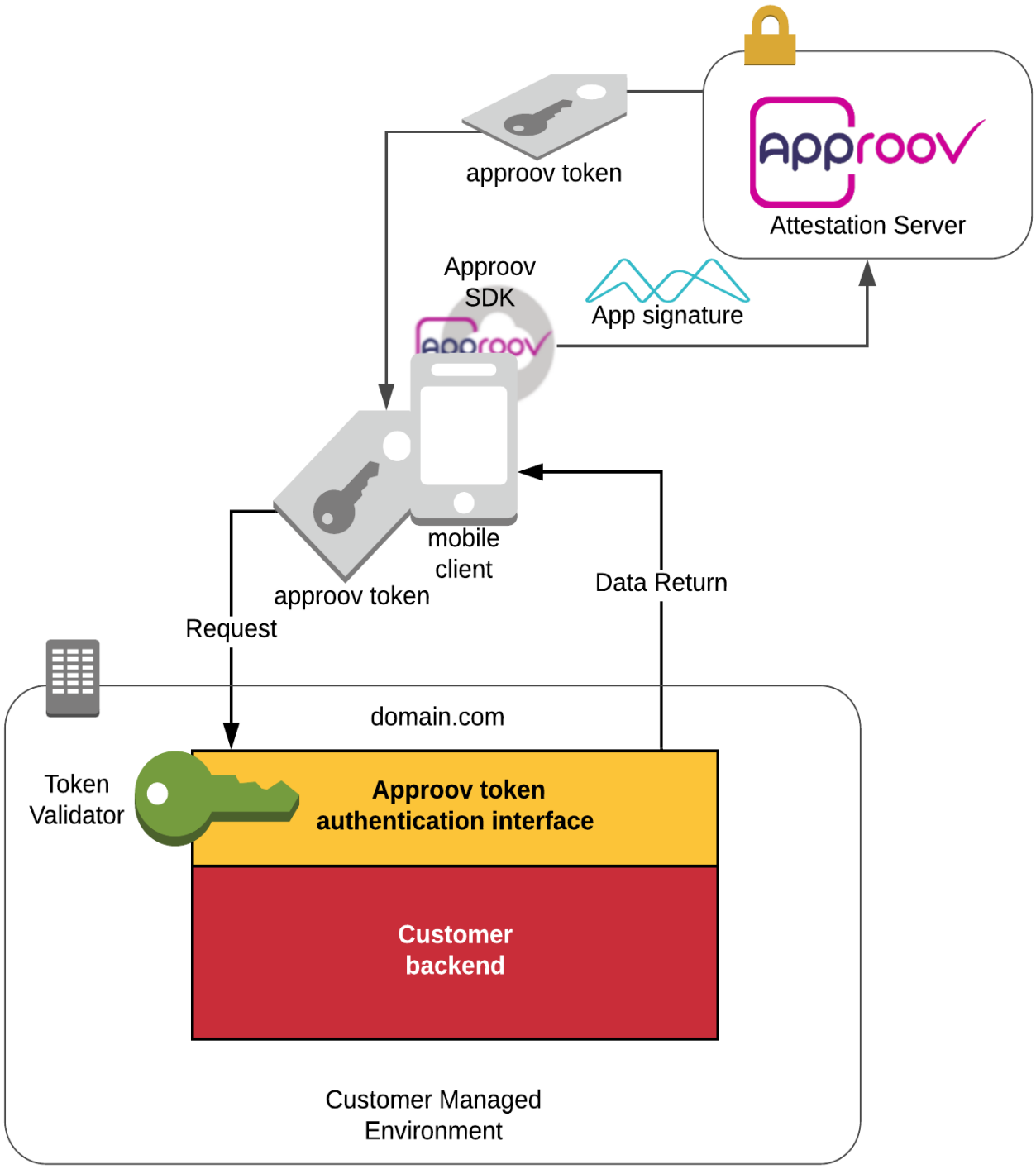
approov

MOBILE APP PROTECTION

Introduction

Approov is a Mobile App security system that seamlessly allows only your genuine mobile apps to authenticate with a protected backend API, allowing your apps access to your protected private data.

Approov has 3 parts :



- A Client SDK that is built into your iOS or Android app, that communicates with an app image attestation server.
- A secure cloud based attestation service which issues tokens to apps containing the Client SDK and that successfully attest.
- Code which you add to your backend API, to validate the Approov token sent with requests from the app.

Integrating Approov into your existing API infrastructure requires your backend API to have the functionality to decode and validate tokens to allow your Mobile Applications, and only your applications, access to your API data.

However, all public requests, from genuine apps, cloned apps, data scrapers and potentially, even denial of service attacks, will be able to at least attempt to connect to your publicly facing API.

What is required is a cost simple effective solution that will independently sit in front of your existing infrastructure, between the client and your backend API, controlling access by validating the Approov token in each request, allowing your apps to seamlessly access your API data, but denying access to all invalid requests.

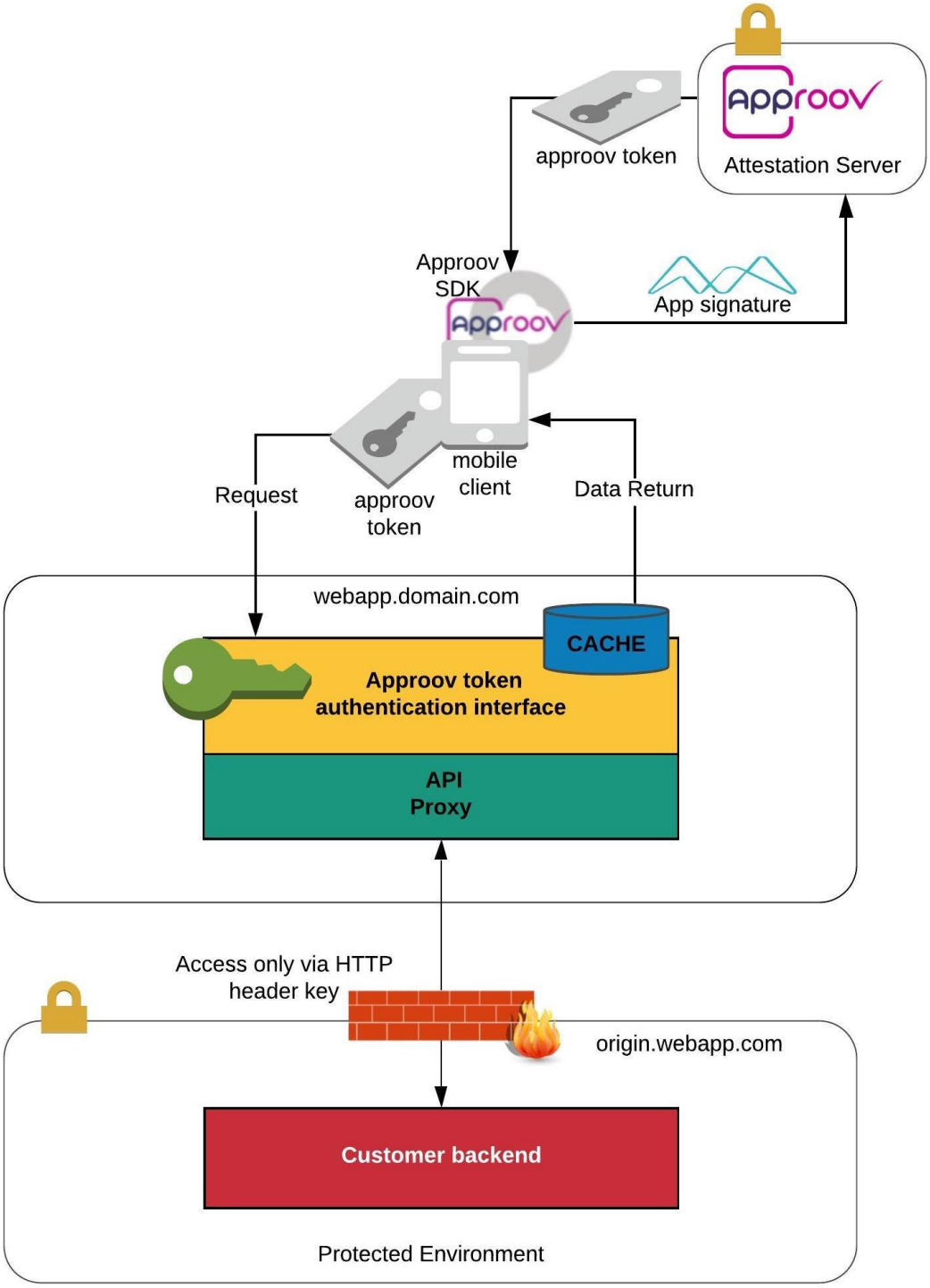
This solution can and should also prevent Distributed Denial of Service (DDoS) attacks, against your API.

The existing backend should only serve content to our cloud based serverless solution.

This paper sets out a simple cost effective serverless cloud based secure reverse proxy that can be built independent of your own infrastructure, that will sit inside a cloud provider, between your mobile app in the public space and your existing API backend.

Approov Serverless API Reverse Proxy

This is an introduction into a cloud based API authentication layer that will securely allow authorized access to your API data, through validating requests using Approov JSON Web Tokens (JWT), this reverse proxy layer can also, optionally, cache tokens and access permissions, as well as API data retrieved from the custom backend origin.



We will explain how to build this solution, using AWS serverless lambda functions, AWS API Gateway, and choosing whether to enable the use of the internal data and token Caches. How to configure transaction logging, and methods to monitor and manage this environment.

We will also formulate example costs, not only concerned with creating this system, but also calculate the expected operational costs per request arriving from the mobile client.

We will set out transactional performance, including expected timings for request handling and data turnaround times.

The following is a reasonably easily created solution that can not only carry out these tasks but can also log requests, both good and bad, and is also capable of generating private long dated tokens for constant access to the same Approov JWT protected endpoints.

AWS API Gateway has been chosen for the number and range of features that we can use, you only need to pay for what you use, technically you can have zero cost for zero use.

There are also many other additional AWS services and features that can be added and configured in the future if so desired.

The Build Process

We will use the AWS Console to build out our API Proxy solution, one step at a time.

I believe it is easier to explain the features and sections of the API Proxy build, if you can see the same detail on your screen as what appears in the document, basically, paint-by-numbers. However, I have also included an introduction to AWS Toolsets like the AWS CLI and AWS Cloudformation for those who might want to take this alternative path, than using the Console.

First we will Build and configure the API Gateway, this will include:

- Configuring API Gateway as a straight through Proxy service, all requests and associated data will be forwarded to the Origin Data source
- Creating and configuring a Custom domain to front face the API
- Creating an AWS TLS HTTPS certificate for this Custom domain
- Configuring Route53 to handle DNS requests to the API via our Custom Domain.

We will then secure access to our API backend, allowing only requests from API Gateway to be processed by our Origin. All public requests for data will have to go through our API Proxy.

To allow secure access into our backend origin API, access can be secured by using one of:

- Custom HTTP header
- AWS ACM generated Client certificate

In the next step, we will create the Authoriser, the lambda function that will validate tokens and return an access policy to API Gateway to either conditionally allow access or Deny access to the requested API Data. Our example Authoriser is written in python.

Next is Cache management, both the Data and the Token caches are optional, if enabled, these caches can speed up request performance of your API.

API Gateway can also front face the API with AWS WAF, I'll discuss if the optional Web Application Firewall is really required.

Next section, AWS Toolsets, these are the tools that can be used outside of the console to create, configure, modify and manage your API Proxy. AWS CLI and CLoudformation are the examples we go over.

Data Logging, includes using Cloudwatch, as well as access logs, can be configured with formats that can allow the data to be used by any existing system that currently consumes your backend origin logs.

Generating Long Dated Tokens, is included here for enabling API access for any monitoring systems, Apps running in debug (privileged) mode, and for our external testing tools that require a long dated fixed token, we will then use this token to test the configuration of the API Proxy.

External Toolsets, these are our remote testing tool sets, I have two examples, one is an application, Postman, the other is a command line interface program called curl. I will demonstrate how to use these tools to test the API Proxy as well as the origin, as we build the API Proxy. These tools will allow us to check if the configuration is correct, these can also be used to troubleshoot issues with the API.

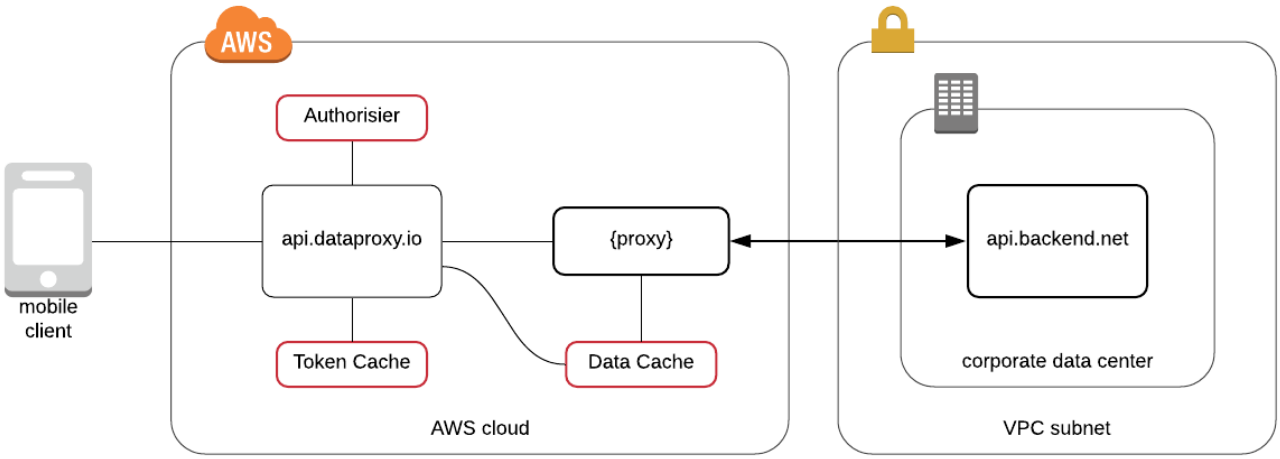
Cost Comparisons, the three main cloud providers, AWS, Google and Azure, are included. There are some big differences here. However each provider does have a different set of services capable of building a Proxy API. Last, is a section on Further Enhancements, namely, Content Delivery Networks, even though these have a fairly limited set of features compared to our “API Proxy”, CDNs can validate a token to provide access to cached data, and can be interfaced with a secured backend.

AWS API GATEWAY

AWS API Gateway is a fully managed, serverless API service, that can secure APIs at any scale. The only costs are the requests that you receive and process, and the amount of data transferred out from AWS, there is no charge for data entering AWS. That's it, there are no servers to manage, no operating systems or application environments to structure, configure or manage, the only configuration required is that of API Gateway, and the creation and implementation of two lambda functions.

How API Gateway works for Approov secured APIs is described in the following diagrams .

Basic overall infrastructure diagram:



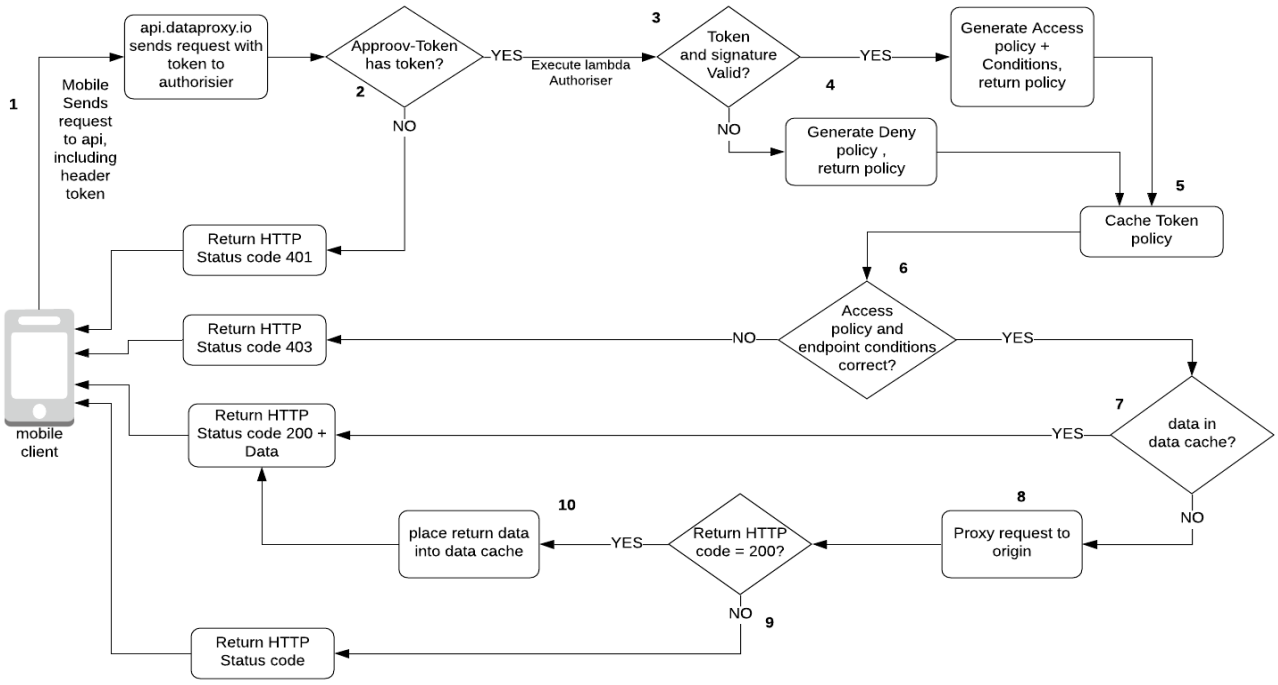
I explain, first, API gateway in the form of api.dataproxy.io, will validate the request, using the authoriser, this is a lambda function that validates and generates the access (or denial) policy for API gateway to service requests to a client requesting data using that token.

If Token cache is enabled, the token cache will be checked for the token and its access policy, this happens before the lambda function is executed. If the token is not in the token cache the authoriser is executed, this validates the token and creates the access policy.

API Gateway then applies the policy to the request.

If the request is valid (Access is allowed and conditions in the policy, if any, are met), and the data cache has been enabled, then the data cache is checked, if the data is in the cache, this will be immediately returned. If the data isn't in the data cache, the request will be forwarded to the origin in the form of api.backend.net, the data is then returned to the requesting client and the data is stored in the data cache, if the data cache has been enabled.

The following diagram is the full detailed workflow how our Proxy will process the request. I will explain in detail how the solution works stage by stage, testing as we go, until completion.



1: The App communicates with the Attestestion server and receives a token, a request to the API is made, with the token included in the HTTP header “Approov-Token” .

2: API gateway checks for the header and value, if there is no token then immediately return a HTTP status 401 (Unauthorisied).

3: If “Approov-Token” Header and value exist, pass these to the authoriser for validation.

4: The Authoriser validates token, if the token is NOT valid, issue a Deny policy, if token valid, issue an Allow Access policy with conditions (time in token has not expired).

5: If token cache is enabled, store the token and the token's access policy.

6: The Policy is then evaluated by API Gateway, if the policy includes an allow statement, and any allow conditions match, then the request is valid, fetch data. If the policy is a deny, or allow access conditions don't match, then return a HTTP status 403 (Forbidden).

7: If the Data cache is enabled, and the data is inside the cache, immediately return this data and an HTTP status 200.

8: Proxy the entire request to the origin API to receive the data.

9: If the origin returns a non 200 return, such as a 404 (Not Found) then return the data (if any), with the HTTP status code. BUT do not cache this return.

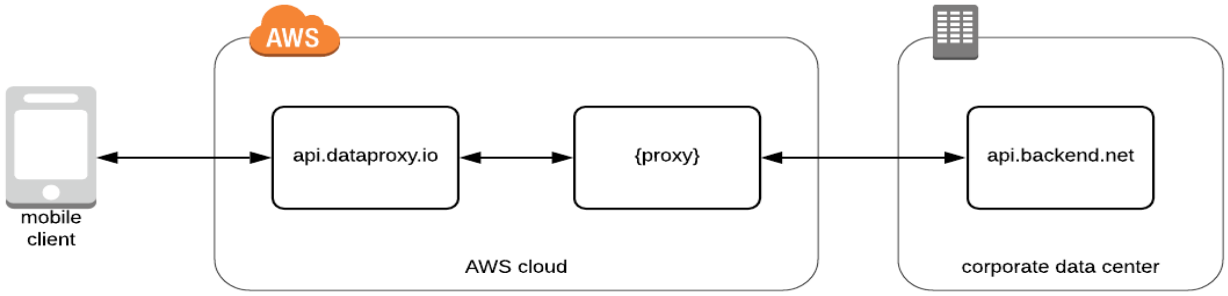
10: Origin returns an HTTP status 200, If the data cache is enabled, API Gateway places the return data into data cache and returns an HTTPs status 200 and the return body (the data) to the calling client.

NOTE: *The authoriser typically should execute and return the access policy in under 50 milliseconds. If data is held inside the data cache the whole request should be processed in under 150 milliseconds. Request to the origin will depend on the connection speed, the physical distance between the AWS region and the origin and of course origin's performance in generating and returning the data. AWS API gateway and the authoriser have a soft limit of 10,000 requests a second. Bursts to 5000 a second can be accommodated, almost immediately.*

DATA PROXY

The first stage of the solution build is to create the Reverse Proxy API, that will take URL requests and traffic going into the AWS publicly facing API then push the same requests, without any URL modifications, out to the real backend API. At this stage, will not enable the data or the token cache, and the integration to the origin will not be secured, we will not yet use the token authorisation, we will first make sure the proxy correctly proxies all requests in the same URL structure, and that the origin accepts these requests and data is returned to the calling client.

Once this stage has been successfully tested, we will fully integrate our origin into our solution and secure it.

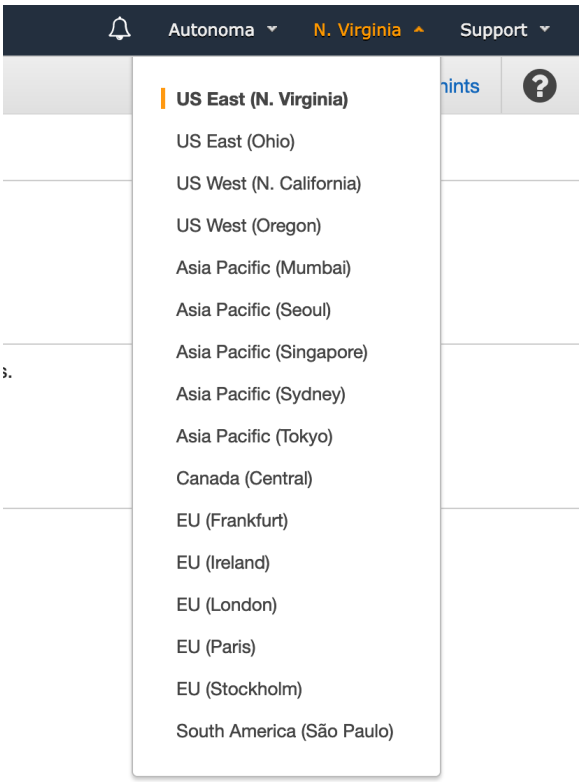


We will start by configuring a new API inside the AWS console, with straight through proxying to the backend data source.

CREATING THE PROXY API

First create the new API, we will call it “Proxy”.

Click on “Get Started”, and select your region, I will be using N. Virginia, as it is close to my origin.

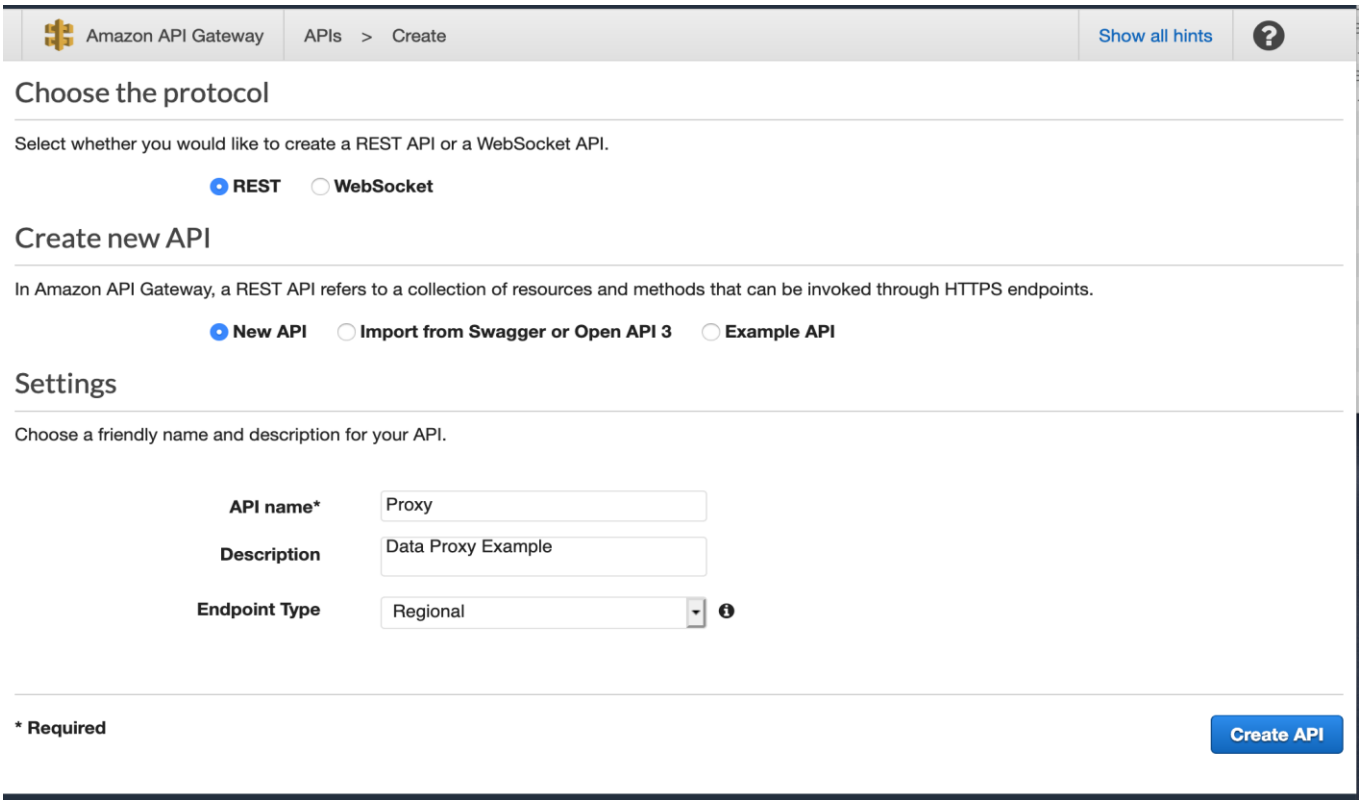


Click on “Get Started”

or “Create API”

Set the Protocol to “REST”, click on “New API”, API Name is “Proxy” give it a description, our API will be a regional one, other options are “Edge Optimised” (uses Cloudfront distribution”) and Private (for APIs only accessible internally within the account).

Choose “regional”. Click “Create API”.

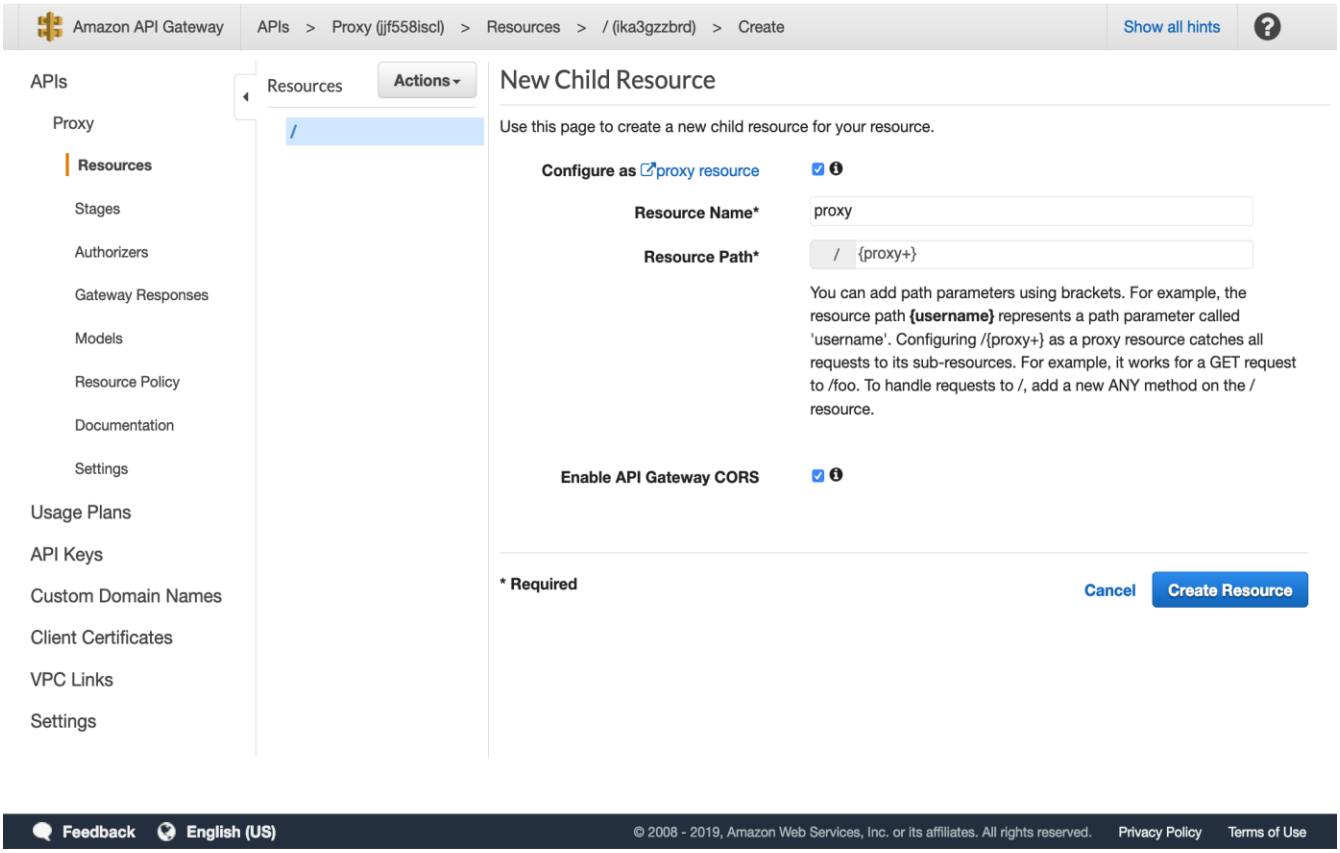


Once saved, you will default to the API’s resource section.

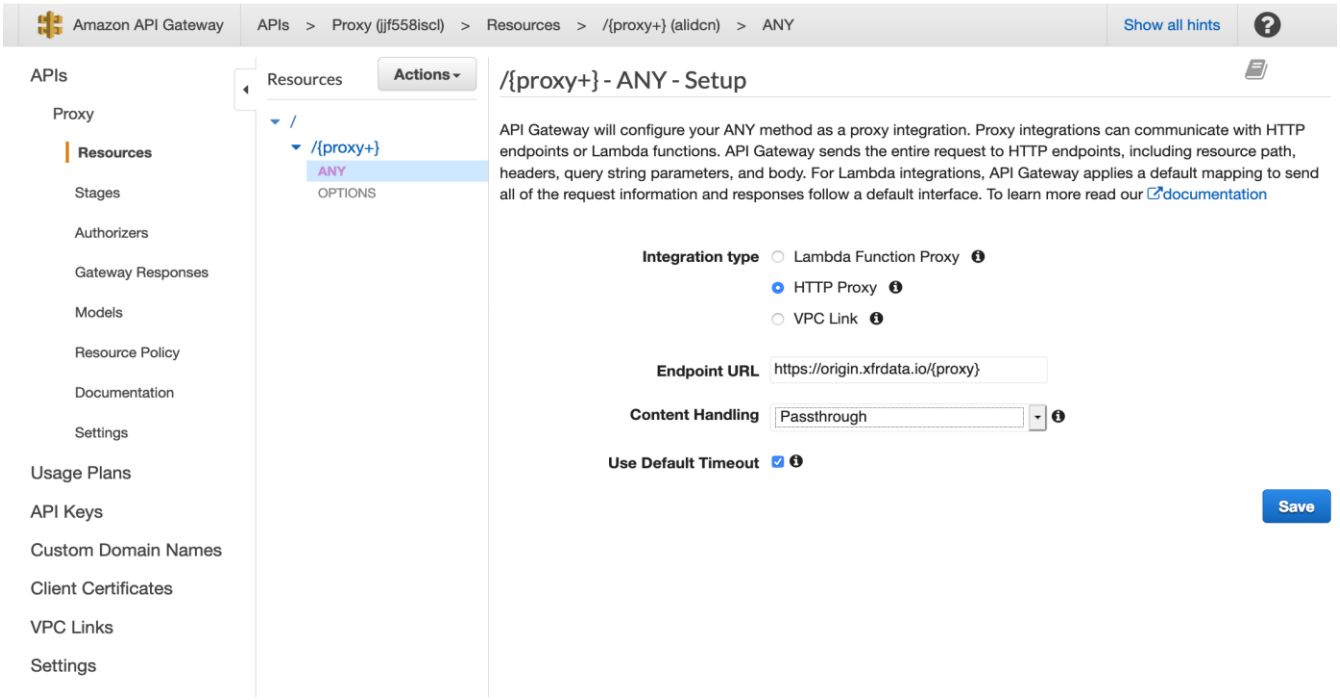
Next, click on the API Root, “/”, then click on Actions -> “Create Resource”.

Click the checkbox “Configure as Proxy resource”. This will autofill the rest of the page automatically.

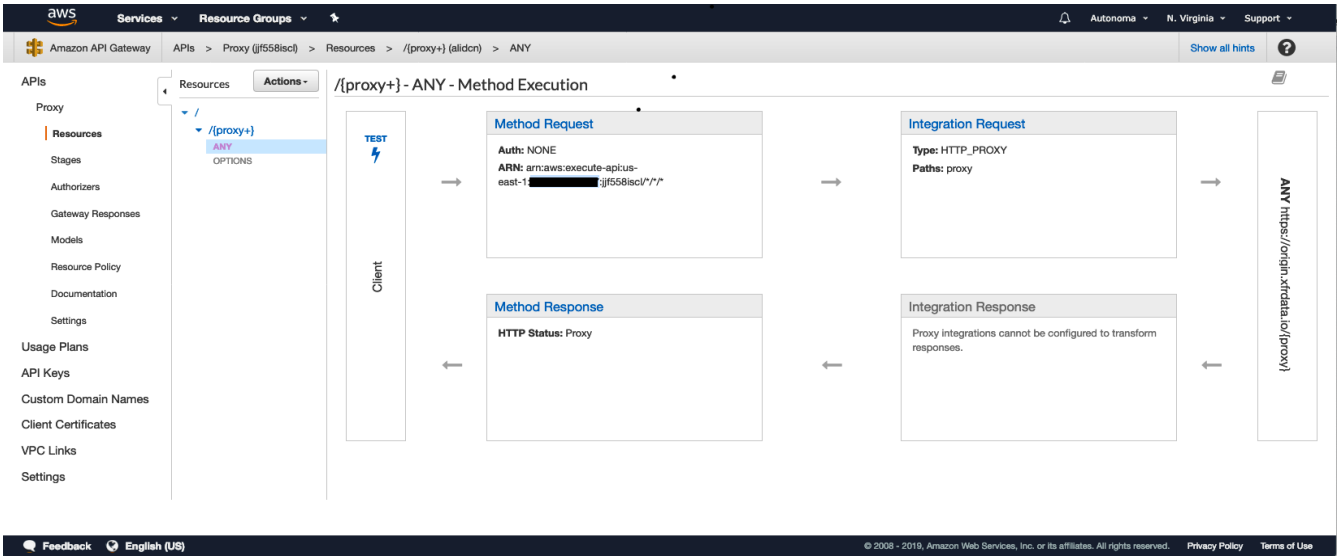
Click on CORS tick box, finally click “Create Resource”:



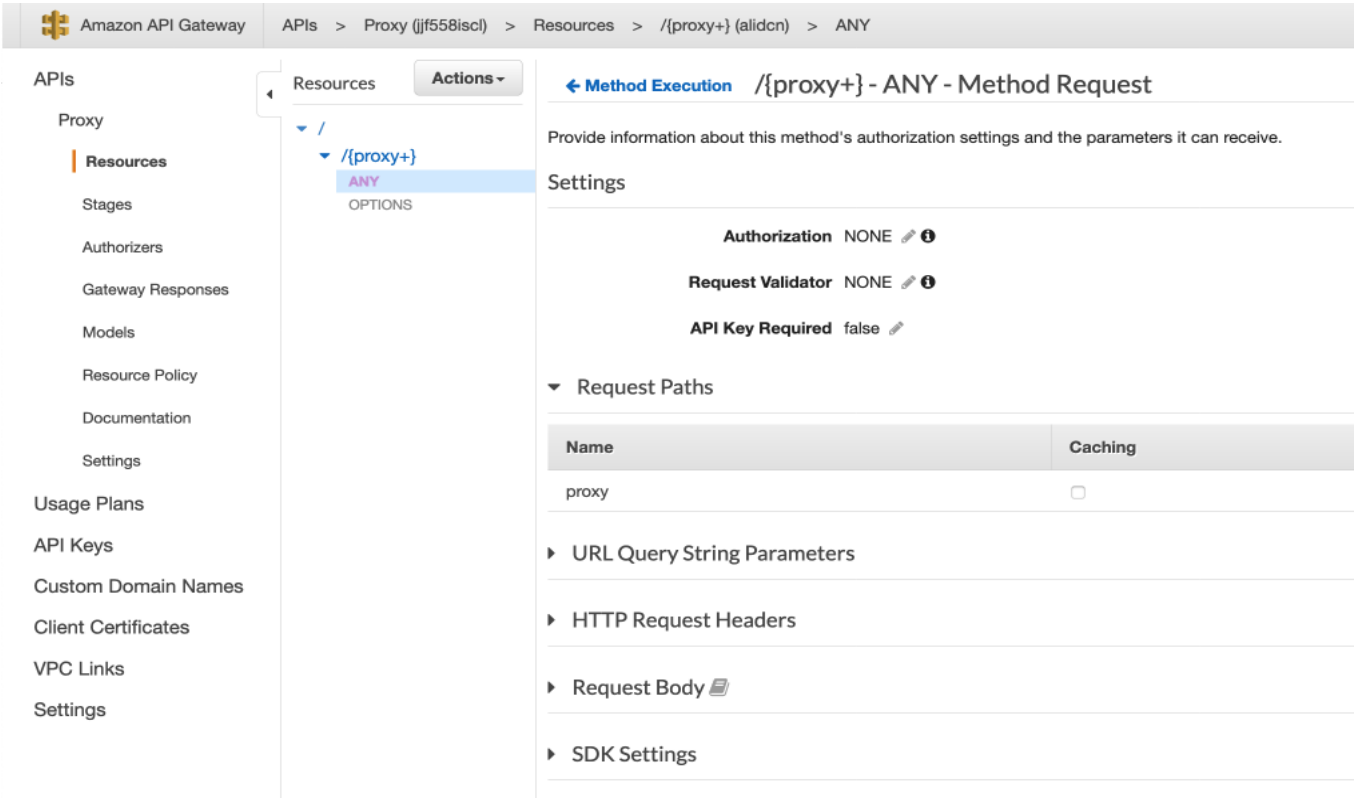
Choose “ANY”, this allows **all** (POST, GET, PUT, DELETE, OPTIONS) HTTP methods to be proxied, then choose HTTP Proxy as the Integration Type. The example endpoint URL is https://origin.xfrdata.io/, and you will need to place {proxy} on the end of the URL like in my example:



Content handling is Passthrough, and I will use the default timeout, which is the maximum allowed 29000ms, and after clicking save, you should be presented with the following screen, where we need to modify the Method request and the Integration Request sections, click on Method Request:



Click on Request Paths and make sure the caching for proxy, tick box is unticked. We do not want caching to be enabled until after the API has been successfully setup.



NOTE: Once the API has been setup, you will need to enable path caching before the API is deployed.

Only once the API has been successfully setup and tested, enable the data cache, this is configured in the Data Cache section.

This also need to be set in “Integration Request” section of the API.

Next Click on “Method execution” (top left) to go back to the previous page.

We now want to test our proxy before we setup the custom domain, TLS Certificates

and Authoriser.

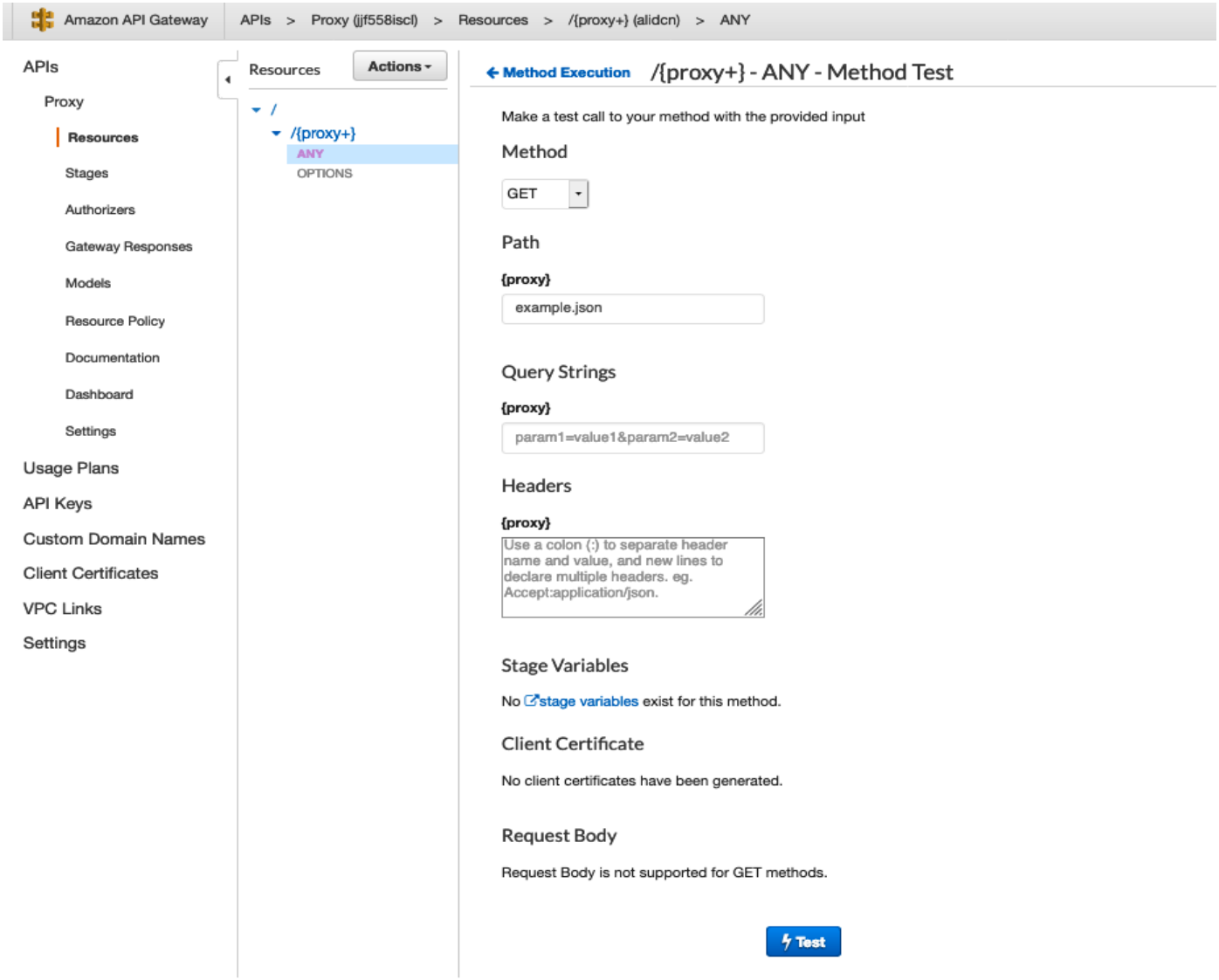
There are two ways to do this, I'd suggest we do both, one will test the resource configuration the other will test the full stage URL.

First click on Proxy, "Resources", then click on the "Test" (with a lightning rod below it) inside the Client Box to the left.

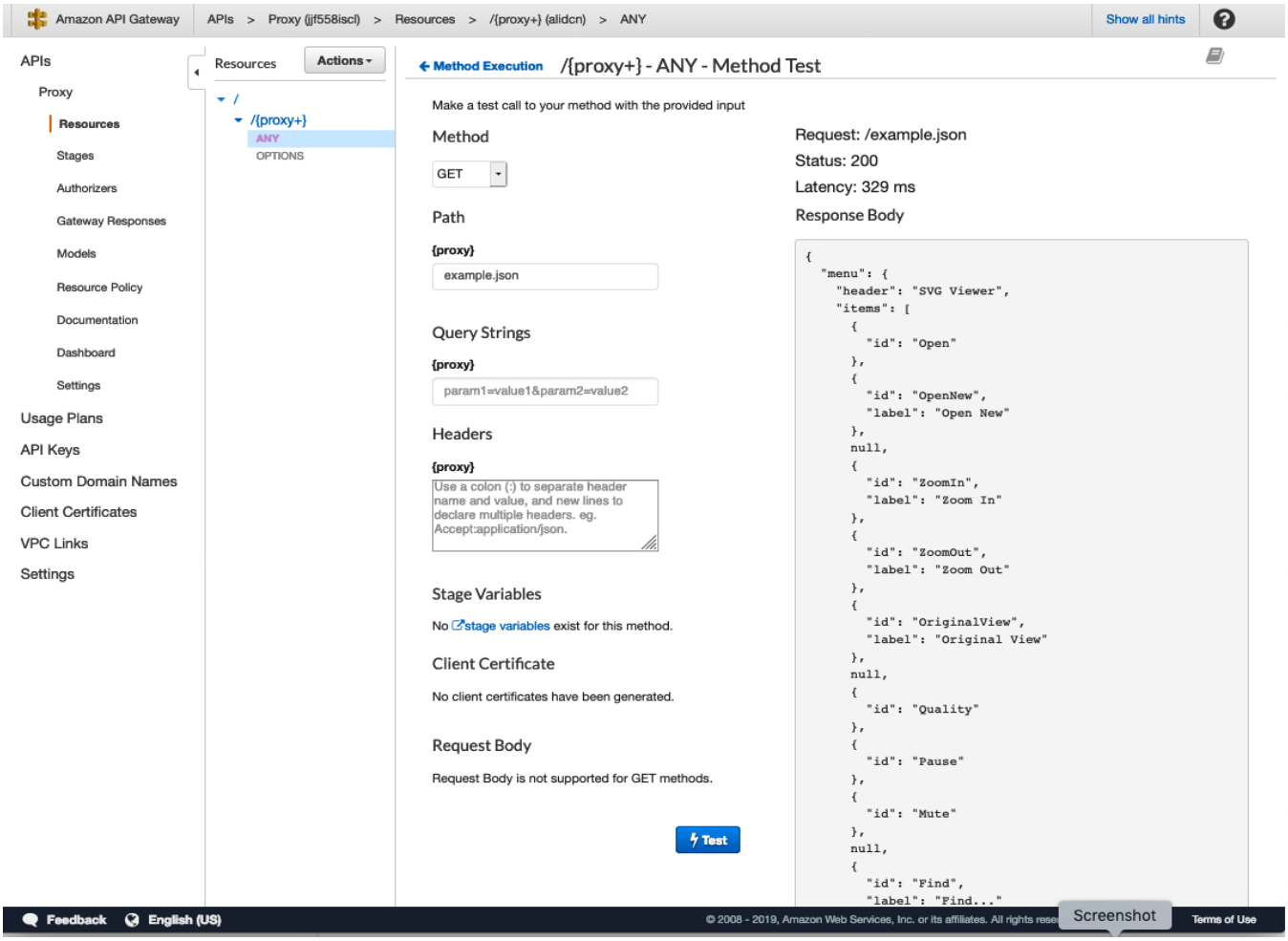
You will get the following page, for us it is just a simple example.json that is located in the root of our origin. Just a simple GET.

I have chosen Method:GET and the file example.json inside the URLbox.

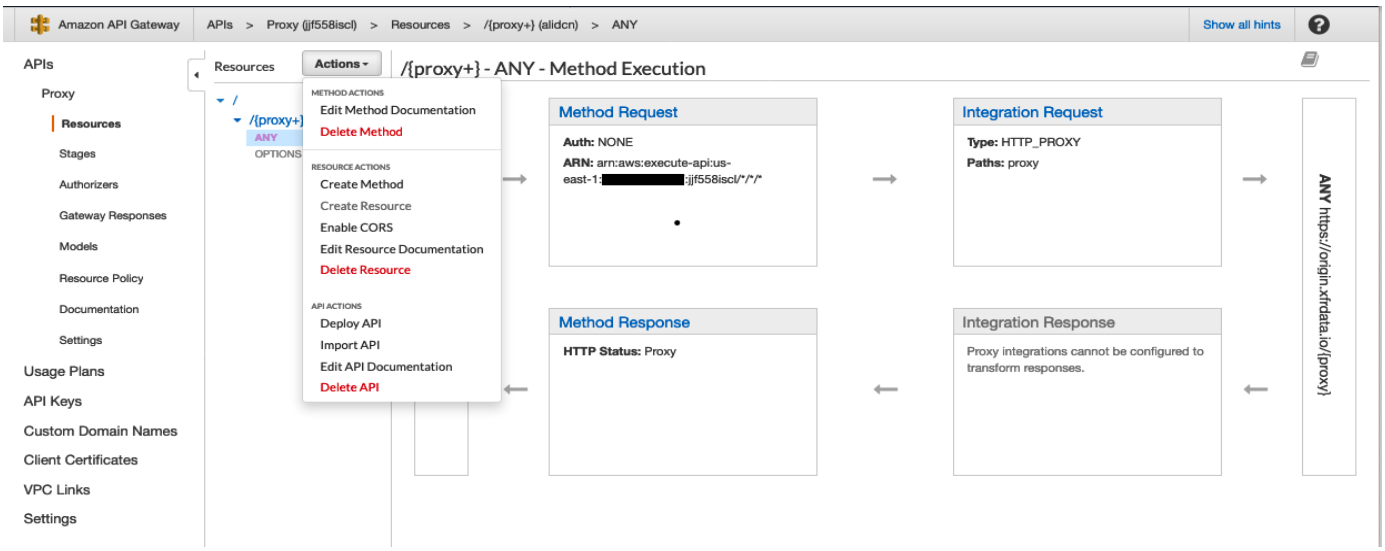
Leave everything else as it is, click test:



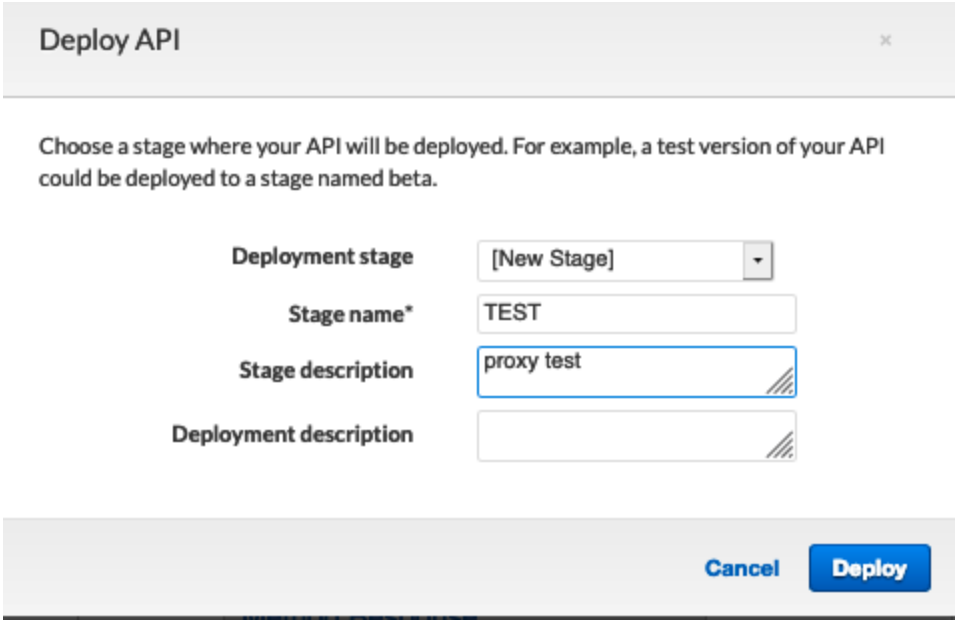
I should get example JSON back from my origin, like so:



We can now deploy the API into a stage and test the external URL. Click on “Actions”, click “Deploy API” (under API Actions).



Choose “New Stage” from the drop down, We will name our stage TEST, give it a description:

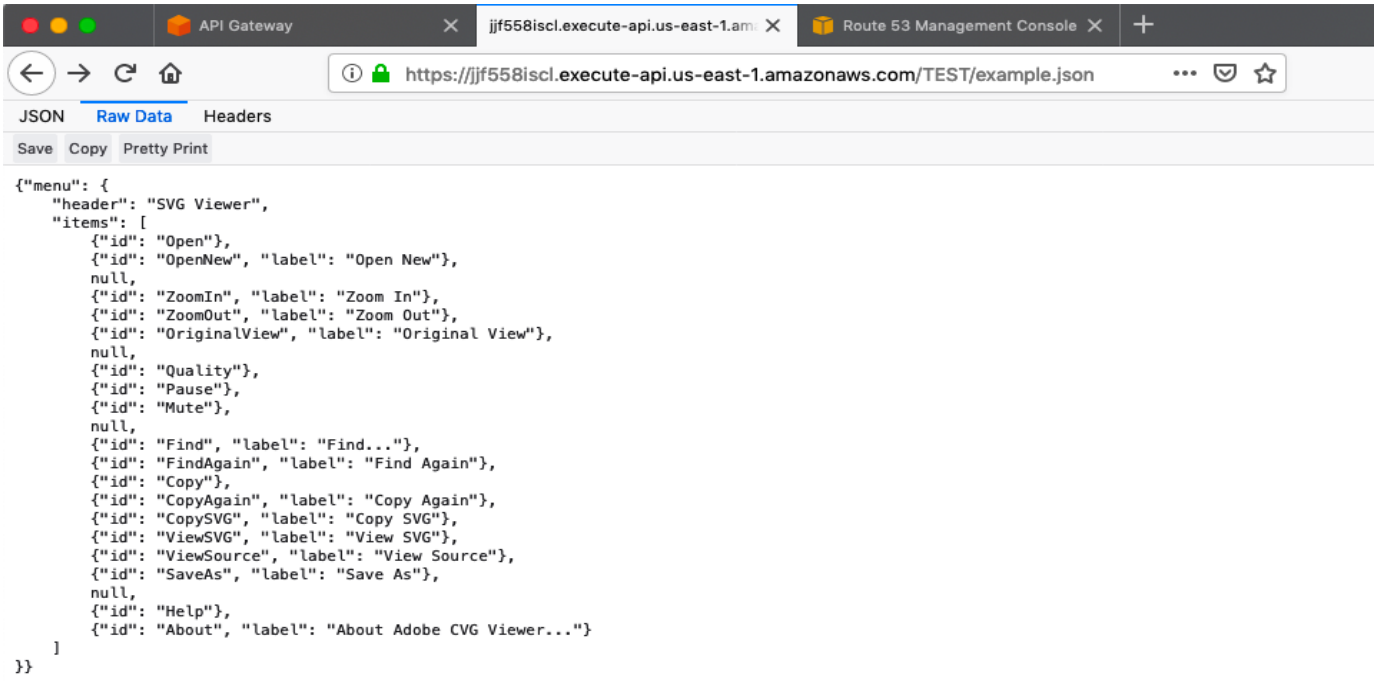


Click Deploy.

The page will refresh inside the new TEST Stage.

The URL of the proxy for the stage is shown at the top of the page.

I have a example.json file in my origin root, this should map to my stage URL, like so…



CUSTOM DOMAINS

You can create a custom domain as an alias to a specific API stage URL.

So instead of having a Public API URL of :

https://jjf558iscl.execute-api.us-east-1.amazonaws.com/Approov/example.json

we can configure our own domain to access a specific stage, a deployed API.

We will use api.dataproxy.io for our custom domain example.

So the above URL becomes : https://api.dataproxy.io/example.json

I will now create a custom domain from my route53 hosted domain dataproxy.io, whose DNS is managed by Route53 nameservers, since this is our domain we will need to generate our own TLS/SSL HTTPs certificate.

We can use the AWS Certificate Manager (ACM) to generate a free TLS certificate.

ACM CERTIFICATE

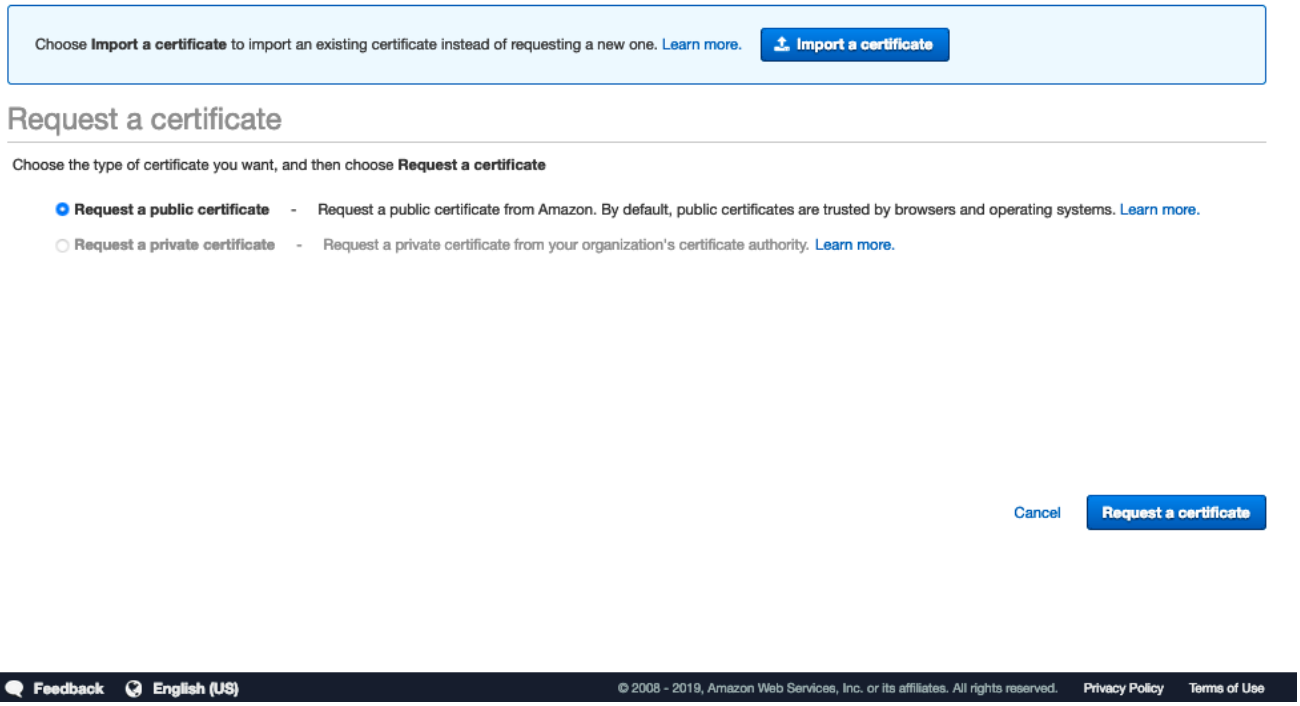
I will first create a ACM wildcard certificate for *.dataproxy.io then create the custom domain api.dataproxy.io inside Route53 and API Gateway, then attach the API stage URL to api.dataproxy.io.

Go to AWS Certificate Manager (ACM), this is a global service, but you will need to choose the region that the API is located within.

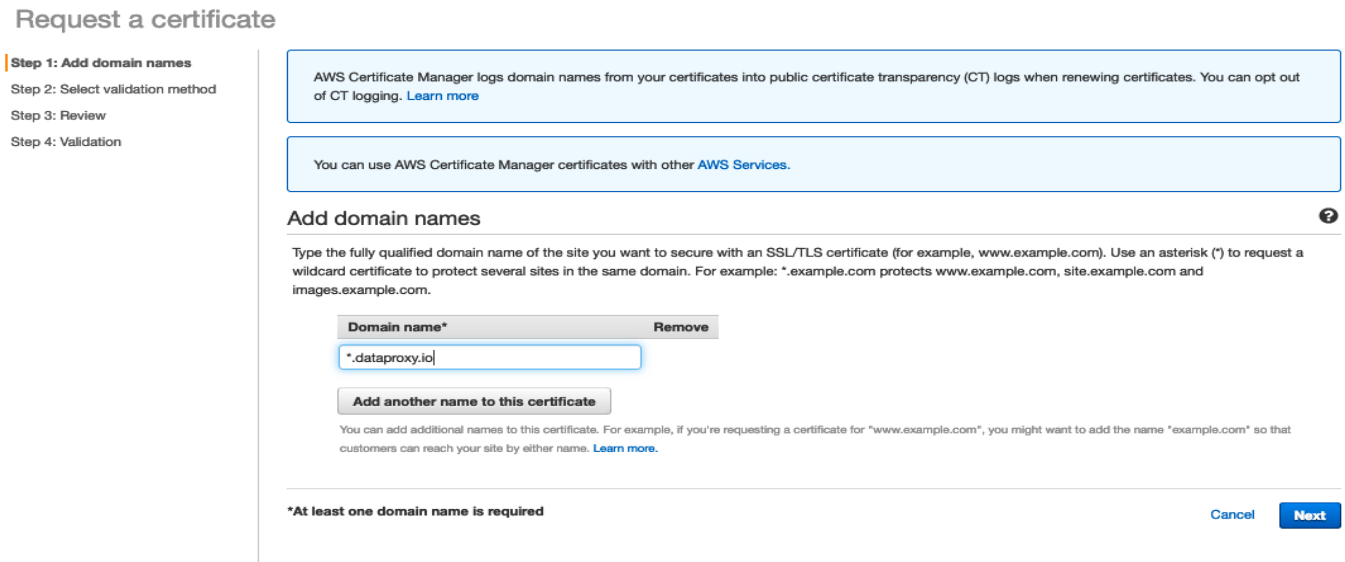
You need to create a certificate in the same region as your API gateway is hosted within, for the custom domain to have the AWS generated TLS certificate attached to it.

Because the dataproxy.io zone in Route53 DNS is managed by AWS, I can just choose to request a free public certificate from AWS. I don't need to import an external Certificate from a public CA.

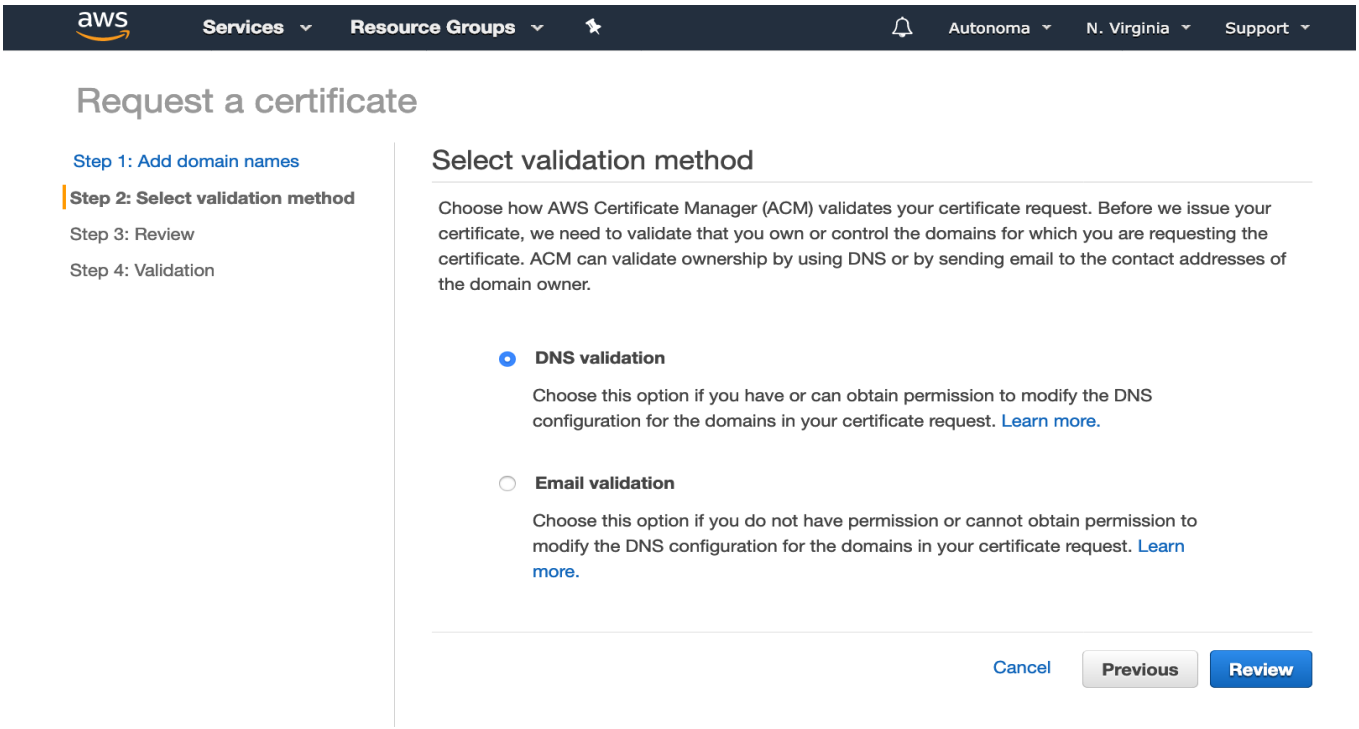
Click on Request Certificate.



I will just make this a wildcard certificate for dataproxy.io

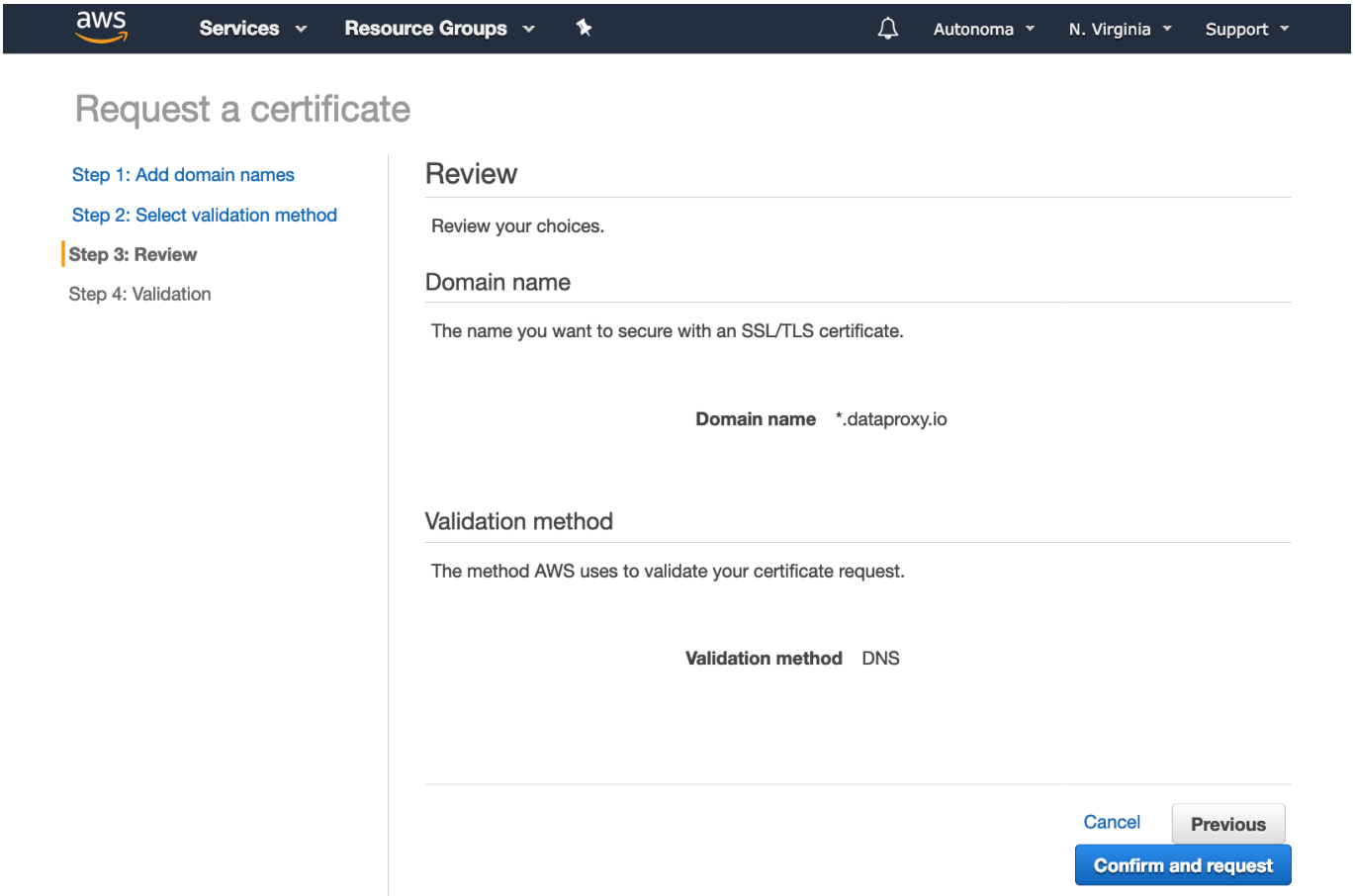


Click Next.

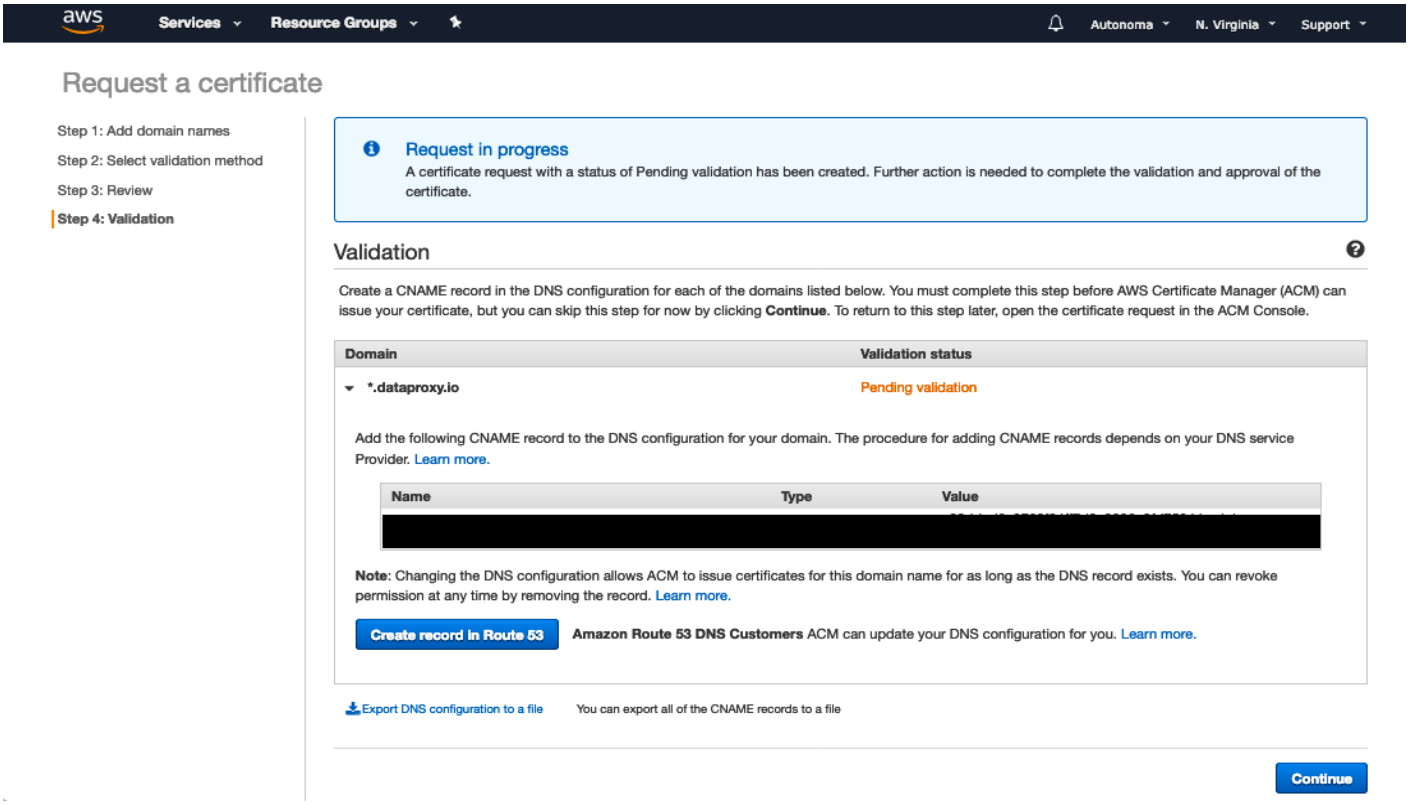


I will choose DNS Validation, it is easier for renewal.

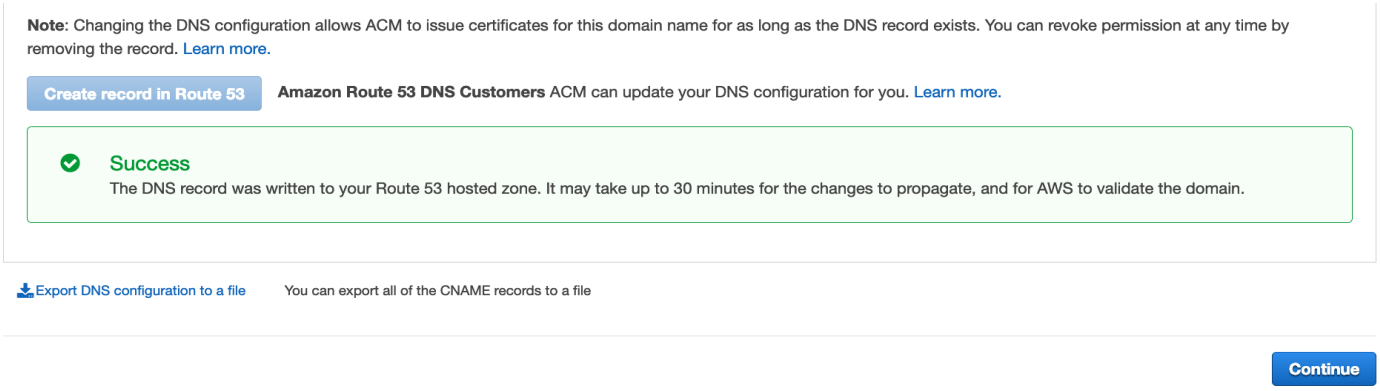
Click Review.



Check the options then Click “Confirm and Request”, this will come up with the validation page.

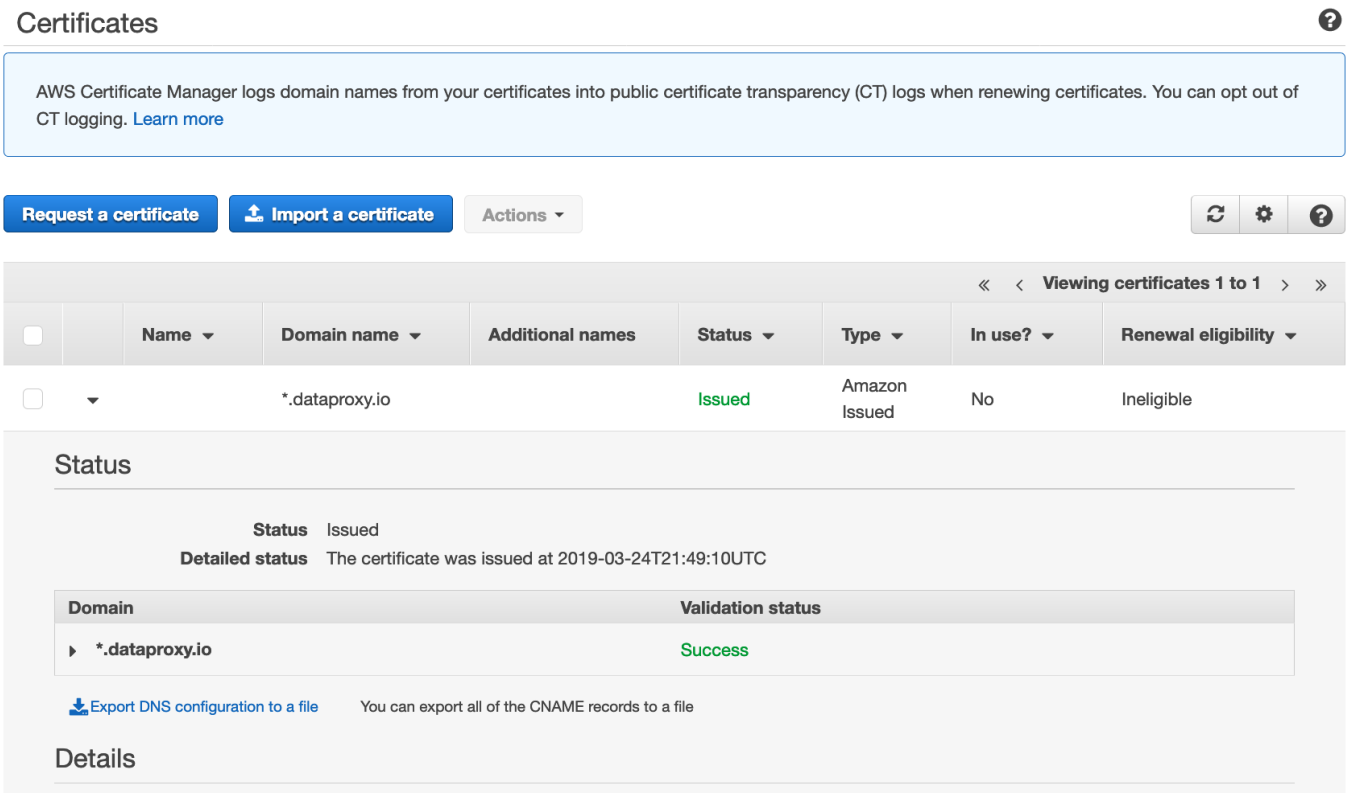


Click Create Record in Route53, this creates a special record that ACM can validate against, externally.



Click Continue.

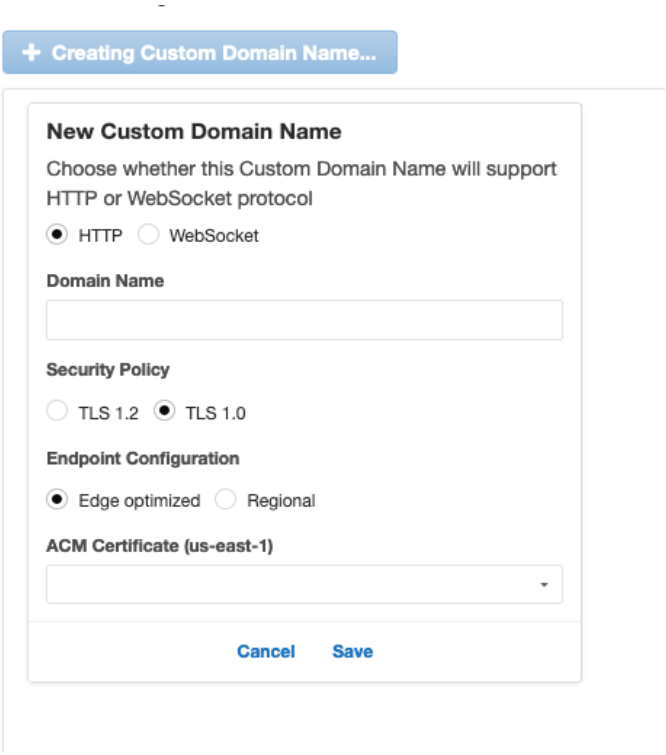
The certificate should now be (or in a few minutes) “issued”.



We can now use this certificate for the custom domain.

CREATING THE CUSTOM DOMAIN

Go back to API Gateway and click on “Custom Domains” then “Create Custom domain”. You will be presented with this page:



For the options: choose “HTTP”

For Security Policy choose TLS1.2

NOTE: *Apple attempted to force IOS Apps to use higher grade SSL/TLS security with regards to HTTPS communications with external resources like API’s, this occurred around late 2016.*

However due to feedback from developers and customers on the work required, and the lack of control over external sites, or on the backend and origin systems to make this upgrade happen, the cutoff date for upgrading external resources with lower grade security including TLS certificates, certificate mismatches and old ciphers, was suspended.

The required level for TLS with regards to this upgrade was TLS 1.2 or higher.

AWS API Gateway custom domain default (at the time of writing), currently is TLS1.0.

TLS1.0 and also SSL have various vulnerabilities such as Poodle, Drown and Beast these can potentially open your API to malicious attack. TLS1.0 also contains the heartbleed implementation bug.

In order to comply with PCI Data Security Standard (DSS), all API’s are required to be TLS1.1 or higher.

I would highly recommend that any API created within AWS should use the TLS1.2 option.

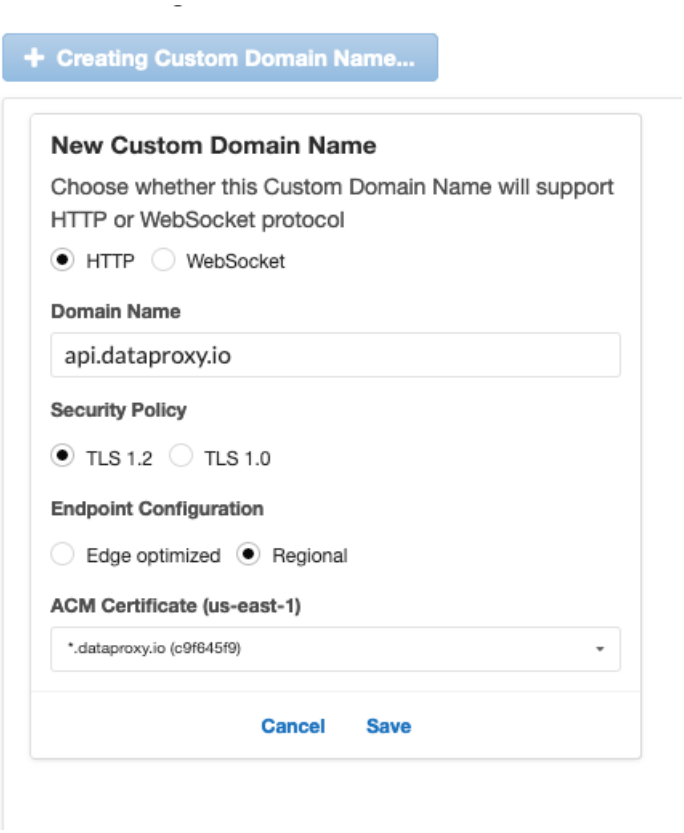
There is a free API TLS compliance tester here.

My Domain Name is “api.dataproxy.io”

Choose TLS1.2 for the Security Policy.

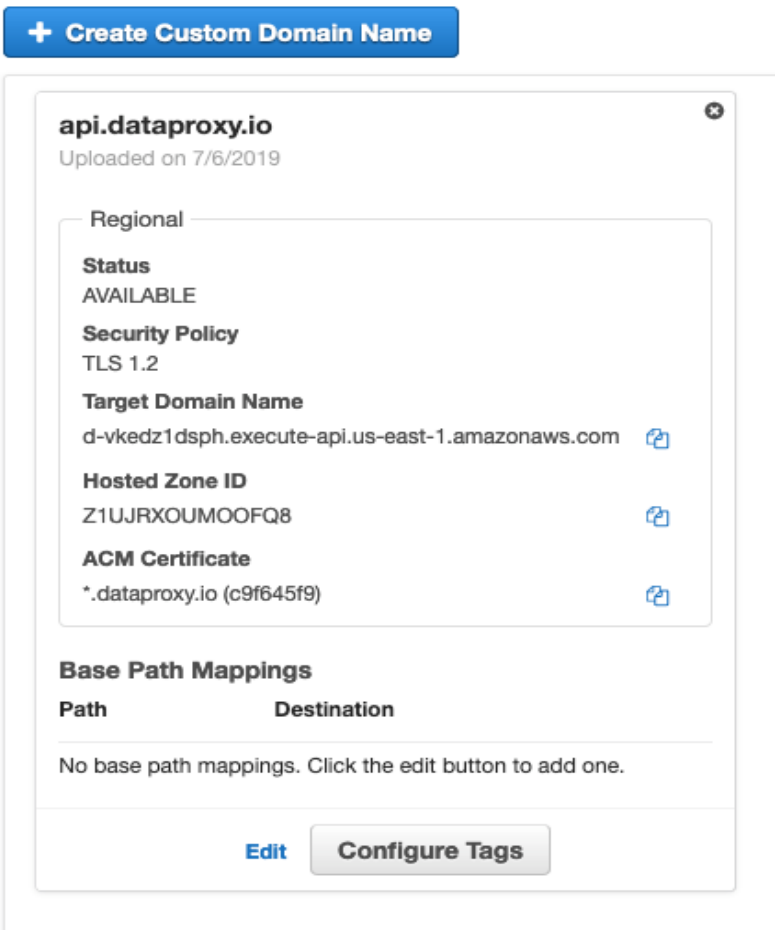
Endpoint Configuration is “Regional”

The ACM wildcard certificate should be in the drop down.



Click save.

After a minute or so the domain will have been created with a target domain that we will use as an ALIAS in Route53 to point our custom domain api.dataproxy.io to a mapped stage of our Proxy API.

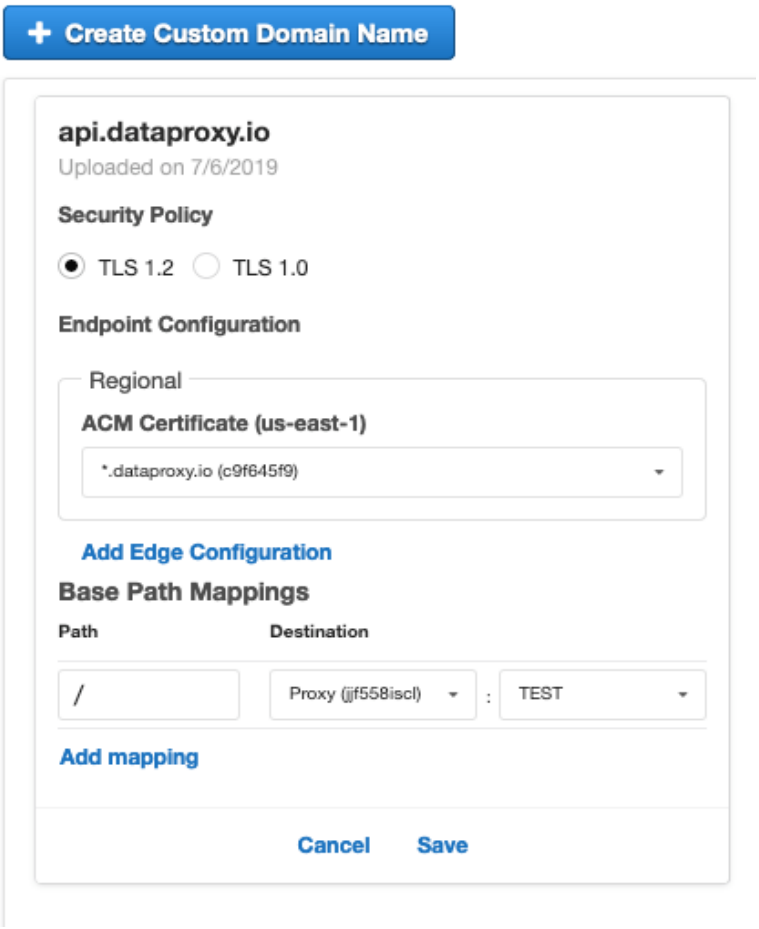


Next click on “show Base Path Mappings”.

This is where we map our api.dataproxy.io to our Proxy API TEST stage.

Click Edit.

Since we are proxy mapping our custom domain to the root of our stage, we only need one map.



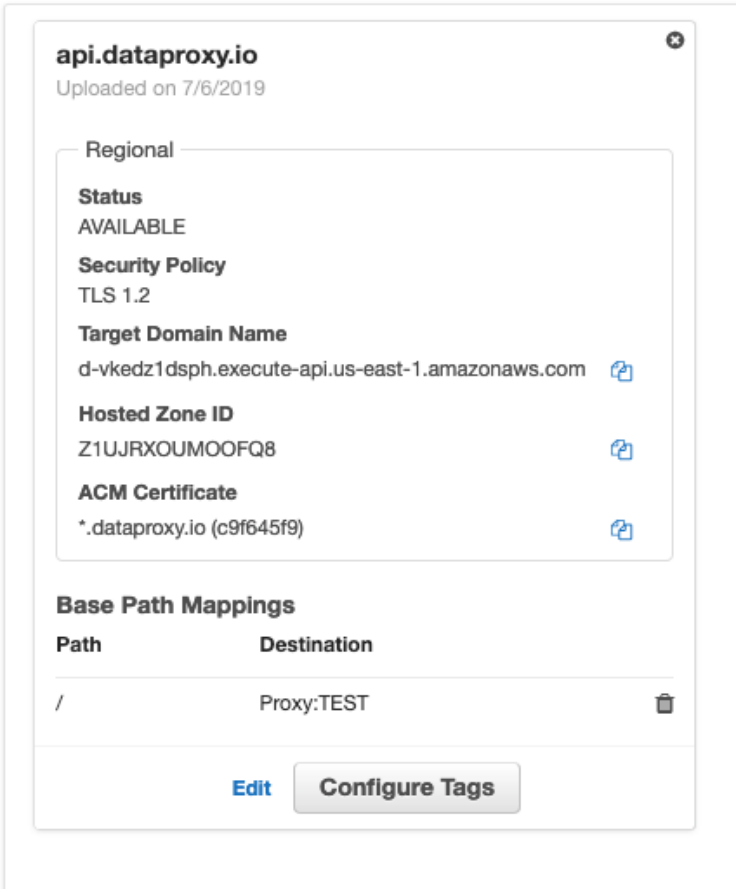
This basically means we are mapping the following dataflow:

api.dataproxy.io/* -> API(Proxy)/TEST/* -> origin.xfrdata.io/*

You can also map other paths to different Stages, and even to different APIs in the same region. But for our root-to-root proxy, this map is all that is required.

Click “Save”.

This will produce the custom domain and make available the “target domain name”.



The “Target Domain Name” is what we need to point our dataproxy.io A Alias record at.

ROUTE53 ALIAS

We now need to create a DNS ALIAS record within Route53, for our custom domain to resolve to our API deployed stage, “TEST”.

An alias is a special custom AWS record, it is similar to a cname but is technically an A Record that maps to the IPv4 ips of the API Gateway Elastic Load Balancer (ELB). Unlike a cname record it doesn't have a DNS time-to-live (TTL), and therefore ips that change within our ELB will be available immediately.

Since we have successfully created the custom domain inside API Gateway, a “Target Domain Name” for our API stage has been created, this becomes the target of our alias.

Go to Route53, hosted zones, click on the domain name to be used.

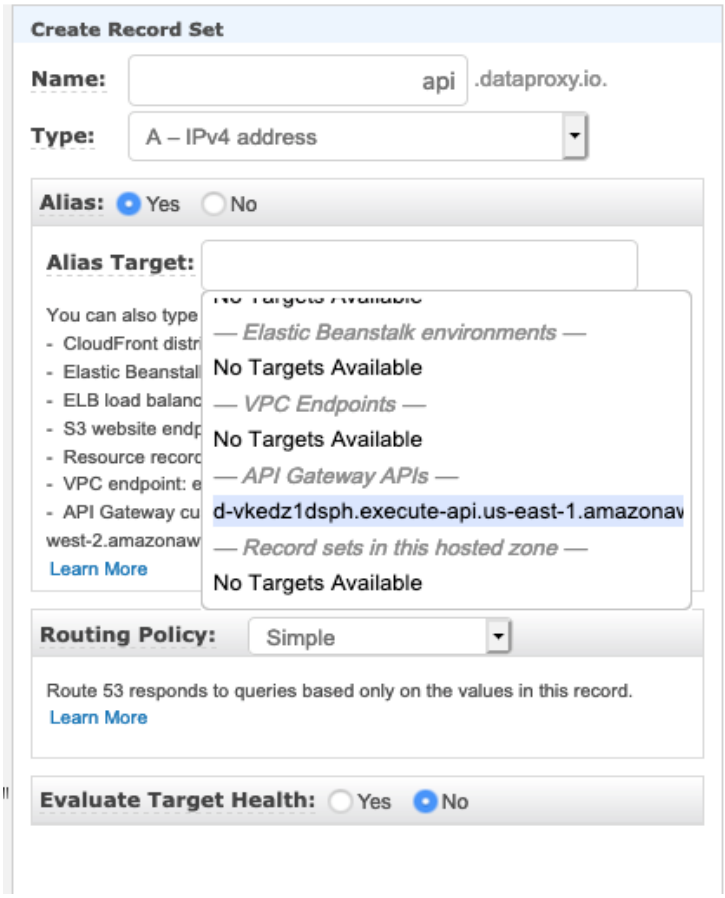
I will use my domain zone dataproxy.io, create an A Alias record.

Name is api.dataproxy.io.

Type is A-IPv4 address.

Alias Yes.

Alias target, scroll down through the options, if the custom domain was successfully created inside the API, the target domain name will be shown in the “API Gateway APIs” section.



Choose the target.

Routing is simple, since our API is Approov protected we need to have “Evaluate Target Health” set to No. Click save.

Now, I will test the configuration to make sure route53 DNS is working.

nslookup not display the target, this is because the record is an alias, and unlike a cname the target is actually a set of publicly addressable resources that the “Target Domain name”points at.

```
gortons@usa1:~$ nslookup api.dataproxy.io
Server:         8.8.8.8
Address:        8.8.8.8#53

Non-authoritative answer:
Name: api.dataproxy.io
Address: 54.156.127.235
Name: api.dataproxy.io
Address: 52.20.97.130
Name: api.dataproxy.io
Address: 54.152.32.14
```

The AWS target IP addresses will constantly change, this is why the DNS resolver will not cache alias data.

TIP: Always use an alias inside route53 for pointing to an AWS alb/elb or any other publicly facing AWS service, instead of a cname. Alias lookups are not charged for, cnames are.

Now test a request, using HTTPs, to our example dataset inside our (as yet, Approov unprotected) custom domain:

```
gortons@usa1:~$ curl https://api.dataproxy.io/example.json
{"menu": {
    "header": "SVG Viewer",
    "items": [
        {"id": "Open"},
        {"id": "OpenNew", "label": "Open New"},
        null,
        {"id": "ZoomIn", "label": "Zoom In"},
        {"id": "ZoomOut", "label": "Zoom Out"},
        {"id": "OriginalView", "label": "Original View"},
        null,
        {"id": "Quality"},
        {"id": "Pause"},
        {"id": "Mute"},
        null,
        {"id": "Find", "label": "Find..."},
        {"id": "FindAgain", "label": "Find Again"},
        {"id": "Copy"},
        {"id": "CopyAgain", "label": "Copy Again"},
        {"id": "CopySVG", "label": "Copy SVG"},
        {"id": "ViewSVG", "label": "View SVG"},
        {"id": "ViewSource", "label": "View Source"},
        {"id": "SaveAs", "label": "Save As"},
        null,
        {"id": "Help"},
        {"id": "About", "label": "About Adobe SVG Viewer..."}
    ]
}}
```

We have completed the main structure of the API proxy, the origin and return, now we need to secure our backend origin so that only our API Proxy can access it. All public requests will now have to come through our API Proxy.

FRONT TO BACK INTEGRATION

We need to secure the backend source of our data, so that only the API gateway can communicate with our backend. There are two methods to integrate API gateway with our backend data source:

- SSL Client certificate
- HTTP Header key

My example will use a nginx origin (apache systems are similar but the configuration syntax might be different.)

I will explain how to use one or the other, probably the simplest is the custom header.

SECURING THE ORIGIN: USING AN API KEY

The Following example sets up an API key HTTP header to pass to the origin when API gateway requests data. We can secure our backend origin in a way that it will only successfully respond to requests from our API gateway proxy, all other sources of requests will be denied.

This is the simplest option to use, it will require no modification once the key has been successfully configured in API gateway and within the backend origin.

We will create our own API key by configuring the header within the method integration section of the API, this section handles the communication with the origin, so that all requests to the origin have the apikey header included within the request.

We will give our header the name “apikey” and a value of ‘HGO05JSN7KWP206HZR2’ (Please do not use this value, generate another unique value for your own API).

My origin is nginx, apache is similar where you set the configuration to look for the header “apikey” with the correct value, if the header does not exist or, the header does but the value is incorrect then the Origin backend API will return a 403 with the message “Unauthorised”.

Create file /etc/nginx/api_keys.conf, and insert the following (using your api key value):

```
if ($http_apikey  !~* 'HGO05JSN7KWP206HZR2') {
        add_header 'Content-Type' 'application/json;charset=UTF-8';
        return 403 '{"message":"Invalid API Key"}';
}
```

Then in the main configuration for your origin (this example is /etc/nginx/site-available/origin.xfrdata.io) include this file at the very beginning of your location section - I have only one, namely the root of the entire API.

```
location / {
        include /etc/nginx/api_keys.conf;
        # First attempt to serve request as file, then
        # as directory, then fall back to displaying a 404.
        try_files $uri $uri/ =404;
    }
```

This will source the api-keys.conf API header check, nginx will return a 403 if the apikey is incorrect or isn't included with the request.

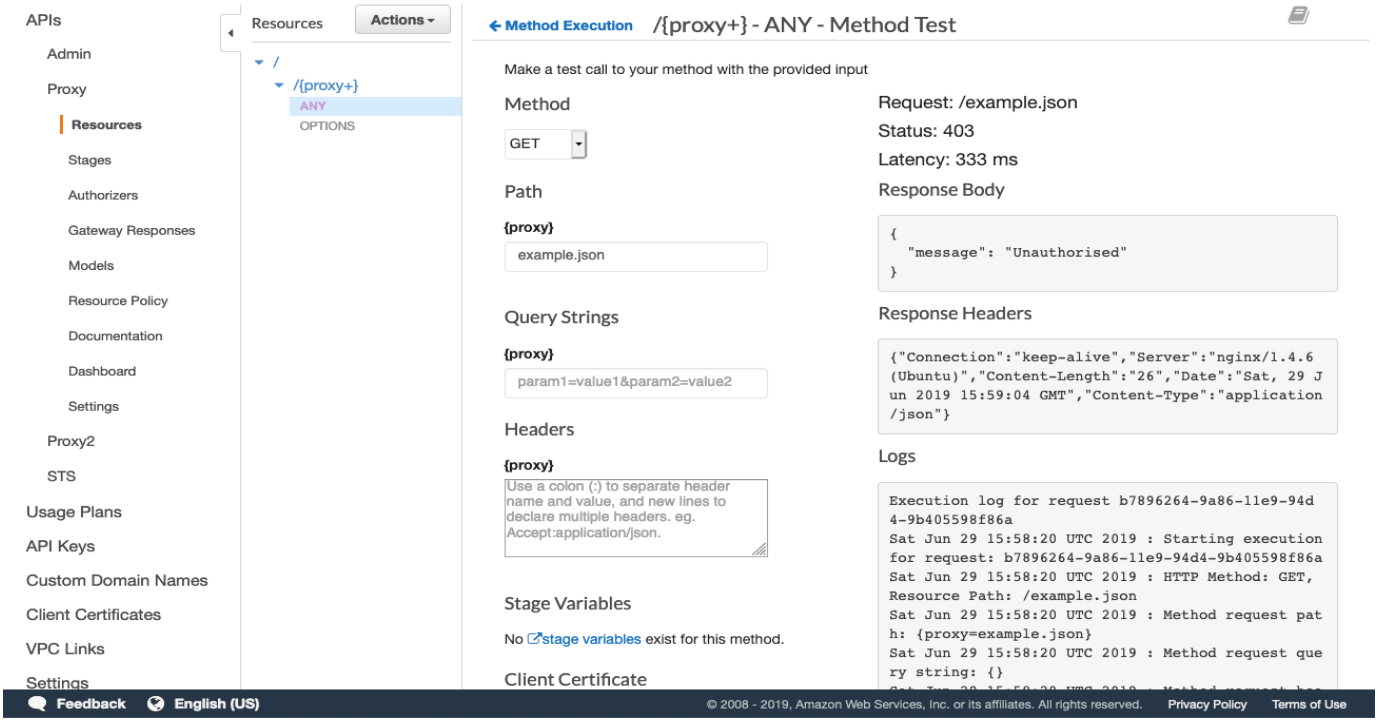
Do a quick check of the configuration, and if OK, reload the nginx configuration .

```
# service nginx configtest
 * Testing nginx configuration                                    [ OK ]
#
# service nginx reload
 * Reloading nginx configuration nginx                           [ OK ]
```

If you have a "FAILED" for the configtest, check your syntax .

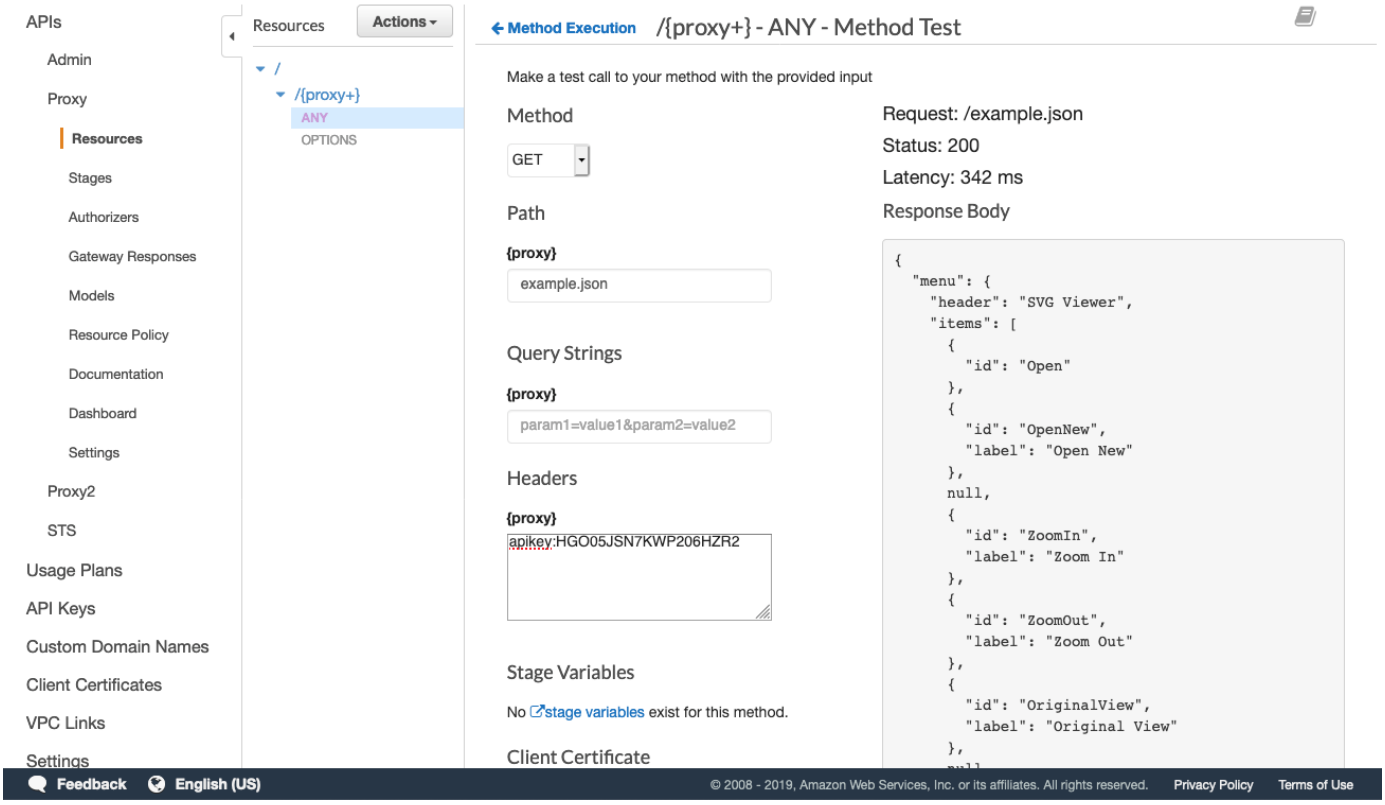
Go to the API->resources->ANY .

click on test, select GET for method, place the "example.json" as our proxied URL into "Path {proxy}", click on the TEST button at the bottom of the page, this will return the 403 with the unauthorised message.



Now, still in the test, type into “Headers {proxy}”, apikey:HGO05JSN7KWP206HZR2.

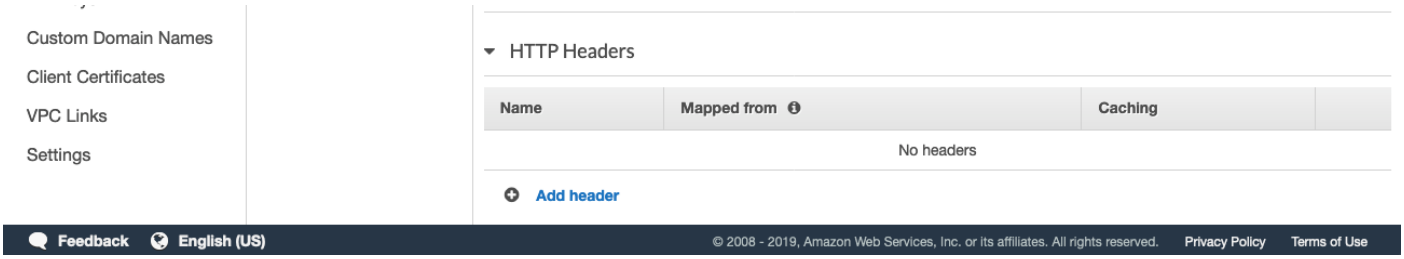
Hit test button, this should return a 200 (request successful) and the correct payload.



This particular test is temporary, the apikey will need to be inserted into the “Method integration” section as a custom header.

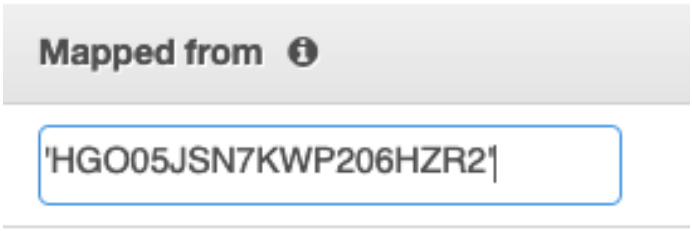
Go to Resources-> Integration Request.

Go down to HTTP Header and open up the drop down.



Click on add header, place “apikey” into Name, and value ‘HGO05JSN7KWP206HZR2’.

because the header string is a static value (and not sourced from method.request) you will need to encapsulate the string with single quotes “ ‘ “

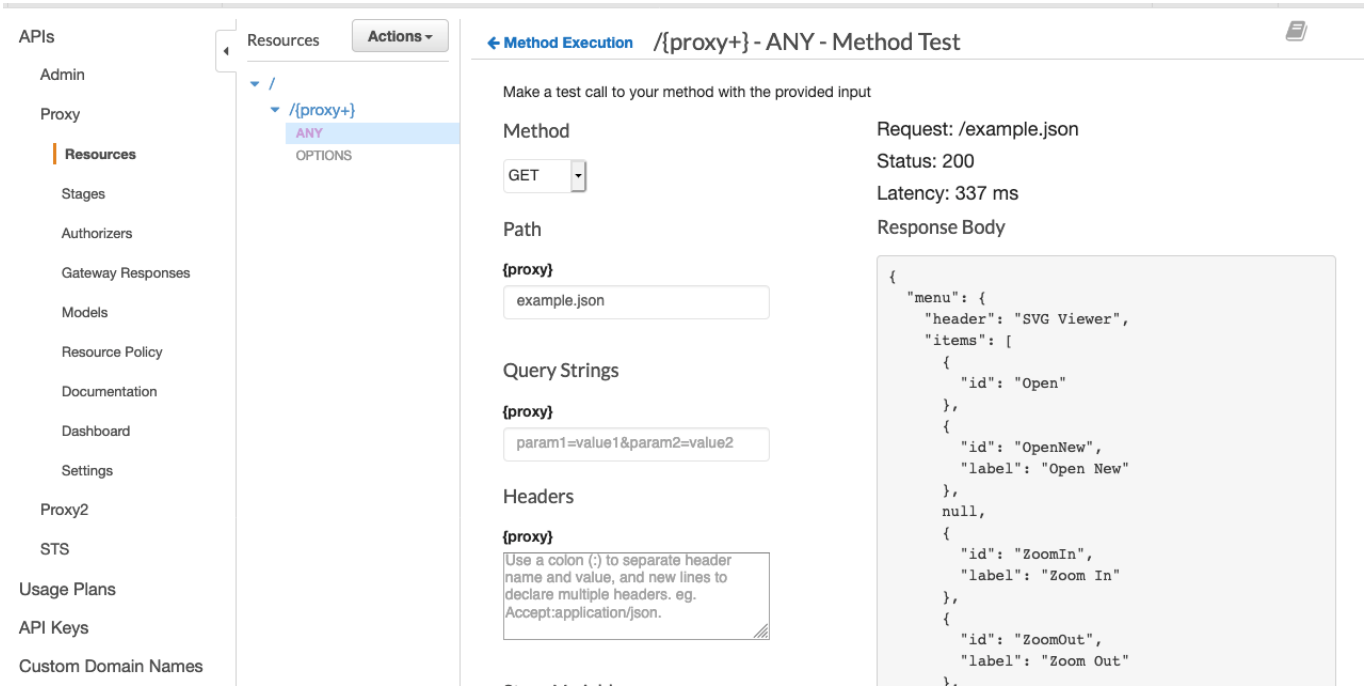


Click on the small create arrow at the right to save this header (example).

Then click on Cache, as this header and its value is permanent.

Now test the access to make sure that the configuration now works without having to manually type in the apikey during the test.

Configure the test like last time except leave the Header section empty, our new header created in the integration Request session will take care of the communication with the Origin.



A Successful 200 return.

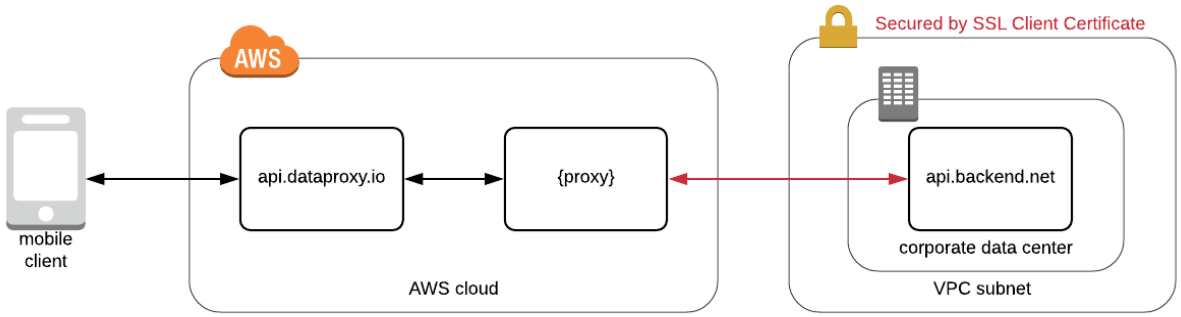
If you use the apikey instead of the client certificate for secure communication with the backend origin, skip “Client Certificate” and go directly to the “Authoriser” section.

SECURING THE ORIGIN: USING A CLIENT CERTIFICATE

Client certificates are attached not to APIs but to API Stages.

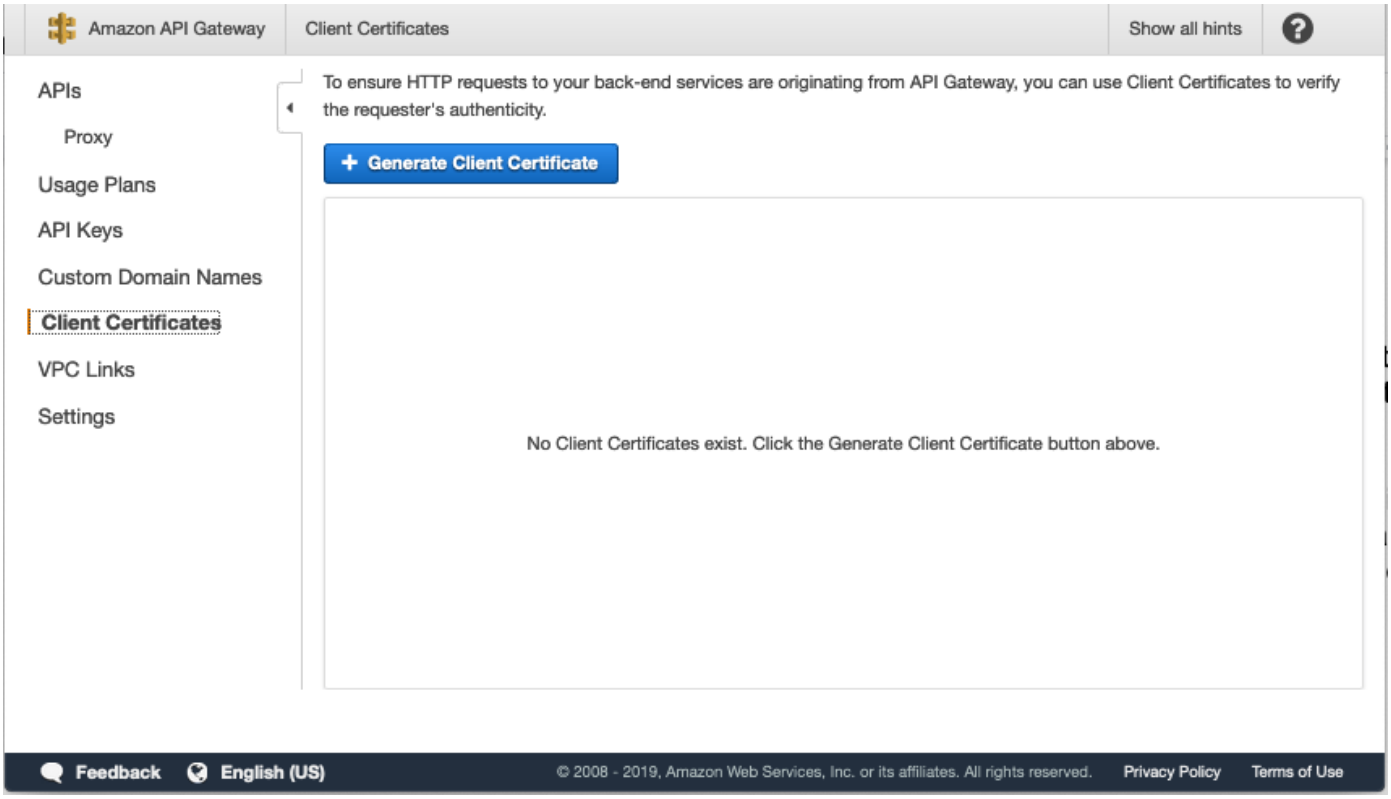
The client certificate requires installation on the origin host, this client certificate also requires renewal and redeployment to the origin, every 12 months.

This is a TLS certificate that basically works in a way similar to ssh public/private keys, it just identifies the calling client.



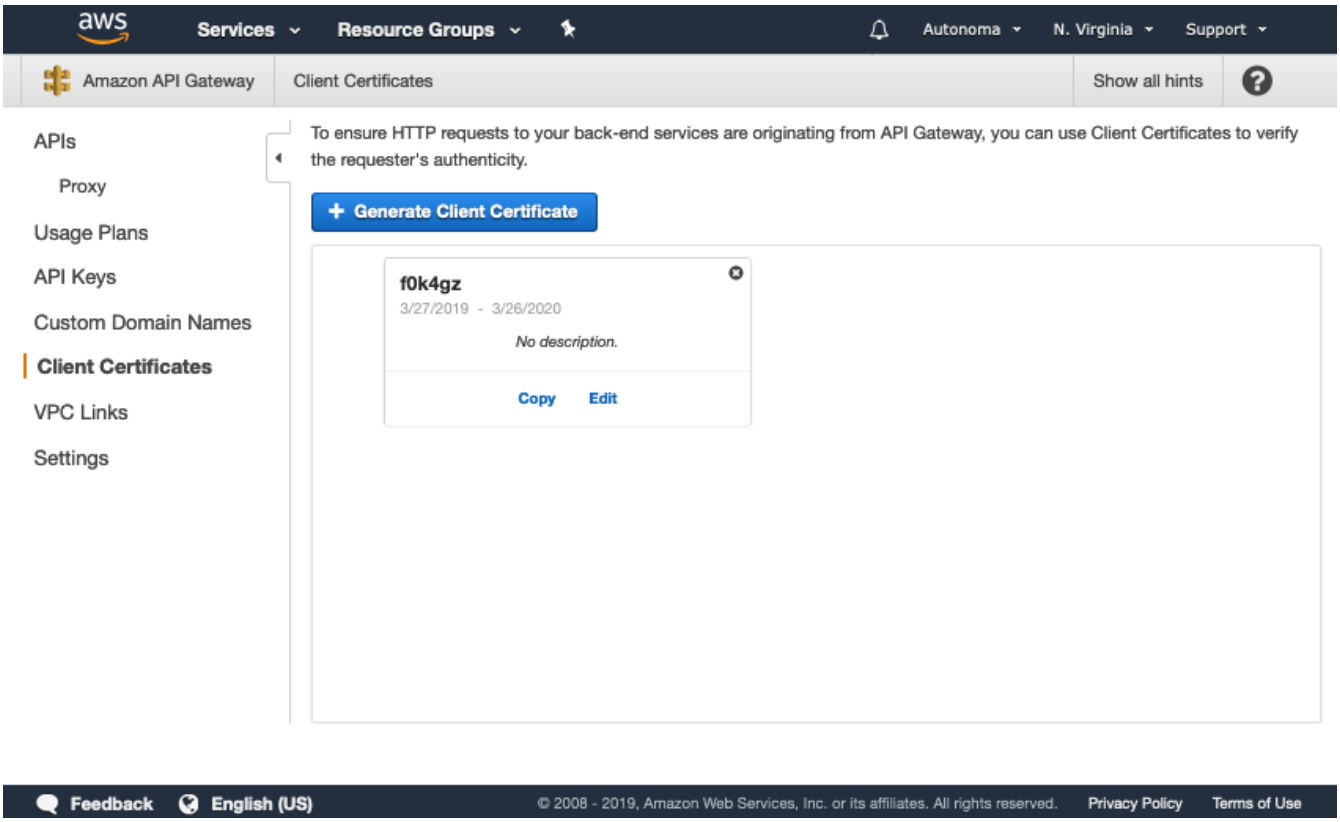
Once the client certificate is generated you will need to install the client certificate on the origin host or hosts, these hosts do however require a valid public certificate authority generate TLS certificate (self-signed Certificates are not currently supported by API Gateway), and for this TLS certificate to have already been successfully configured on all origin hosts before you can install and use the client certificate for the API Proxy to successfully communicate with the backend origin.

So, first, within AWS API Gateway, go to section “Client Certificates.”



Click on “Generate Client Certificate”.

This will generate the certificate with a certificate id, and a validation time frame.



Next click on “Edit” and give the certificate a description, for example what API is used, and what origin.

Click “Save” then click on “Copy”.

This will save the public certificate to your clipboard.

Now you simply, open an editor and paste in the certificate, this is the public key that you will need to install on the origin to allow only proxy API to access the origin.

INSTALLING THE CLIENT CERTIFICATE

My origin is an nginx instance, origin.xfrdata.io. It has a simple configuration for just one location on the server.

Once the certificate is installed, only requests from clients using this certificate and holding the private certificate (namely AWS API Proxy) will be able to successfully access all resources on the server, all other requests will return a 403 (Forbidden) HTTP status code.

I will save my key into a file called /etc/nginx/client_certs/dataproxy.crt

I now need to configure nginx.conf with the following 2 parameters:

```
 # client certificate
   ssl_client_certificate /etc/nginx/client_certs/dataproxy.crt;
   #Make verification compulsory    #IE: So we can send a 403 error to requests that fail authentication
   ssl_verify_client on;
```

Next, set the certificate check within the path of the nginx root location, at the very beginning. If the request doesn't pass the certificate check, return a 403.

```
 # within the root of our origin path, before anything else...
  location / {
    # check that the request has sent correct client certificate.
    # if client-side certificate authentication fails..
    # return a 403 to the client
    if ($ssl_client_verify != SUCCESS) {
      return 403;
    }
```

This is an additional bit of information on my simple origin nginx configuration.

My complete nginx origin server configuration is as follows:

```
server {

        root /var/www/origin.xfrdata.io/htdocs;
        index index.php index.html index.htm;

        # Make site accessible from http://localhost/
        server_name origin.xfrdata.io;

        location / {
                # check that the request also contains the aws dataproxy client certificate.
                # and, if client-side certificate authentication fails, return a 403 to the client
                if ($ssl_client_verify != SUCCESS) {
                    return 403;
                }
                # First attempt to serve request as file, then
                # as directory, then fall back to displaying a 404.
```

```
            try_files $uri $uri/ =404;
            # Uncomment to enable naxsi on this location
            # include /etc/nginx/naxsi.rules
        }
    listen 443 ssl; # managed by Certbot
    ssl_certificate /etc/letsencrypt/live/origin.xfrdata.io/fullchain.pem; # managed by Certbot
    ssl_certificate_key /etc/letsencrypt/live/origin.xfrdata.io/privkey.pem; # managed by Certbot
    include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot

  # aws generated client certificate
    ssl_client_certificate /etc/nginx/client_certs/dataproxy.crt;
    #If we make verification optional then we can return a 403 error for all requests that fail authentication
    ssl_verify_client optional;
}

server {
    if ($host = origin.xfrdata.io) {
        return 301 https://$host$request_uri;
    } # managed by Certbot


        listen 80;
        server_name origin.xfrdata.io;
    return 404; # managed by Certbot
}
```

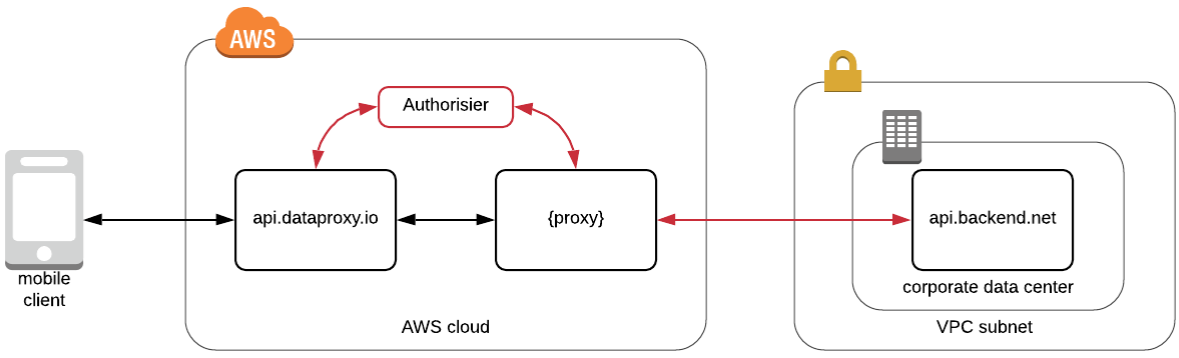
The red text is the newly added certificate configuration.

The Blue shows the existing valid TLSv1.2 public CA certificate, you will need to create this and successfully install it, on the Origin host, for the client certificate to successfully work.

I use certbot, a certificate renewal and management tool, to control the installation and renewals of a free 3 month rolling LetsEncrypt TLS certificate for my origin nginx instance.

Refer to AWS Client Certificate Rotation for more details on renewing and redeploying client side certificates.

AUTHORISER



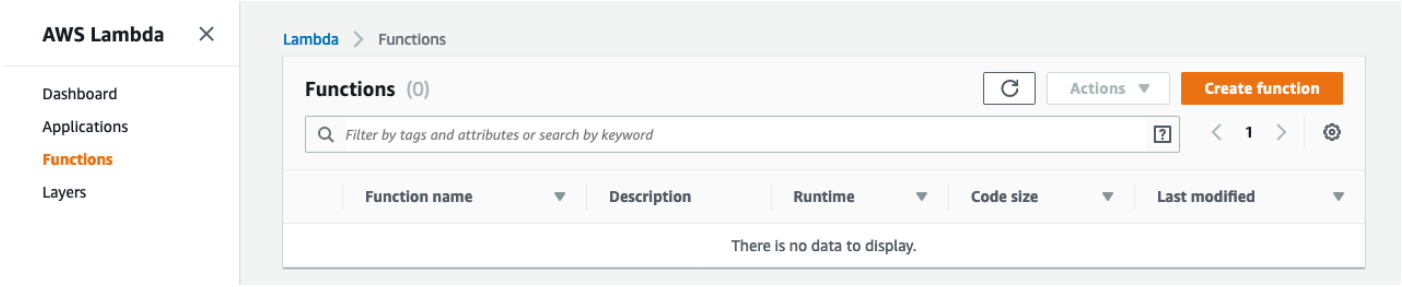
The Authoriser is part of the Method Request section within API Gateway.

This takes the Approov JWT (JSON Web Token) that is sent by the client request, and validates the JWT by generating a signature of the header and payload body and comparing it to the JWT's signature, if the signature is a match AND the conditions inside the payload body are true, then the request is authorisied.

First we need to upload the function package made up of Python code.

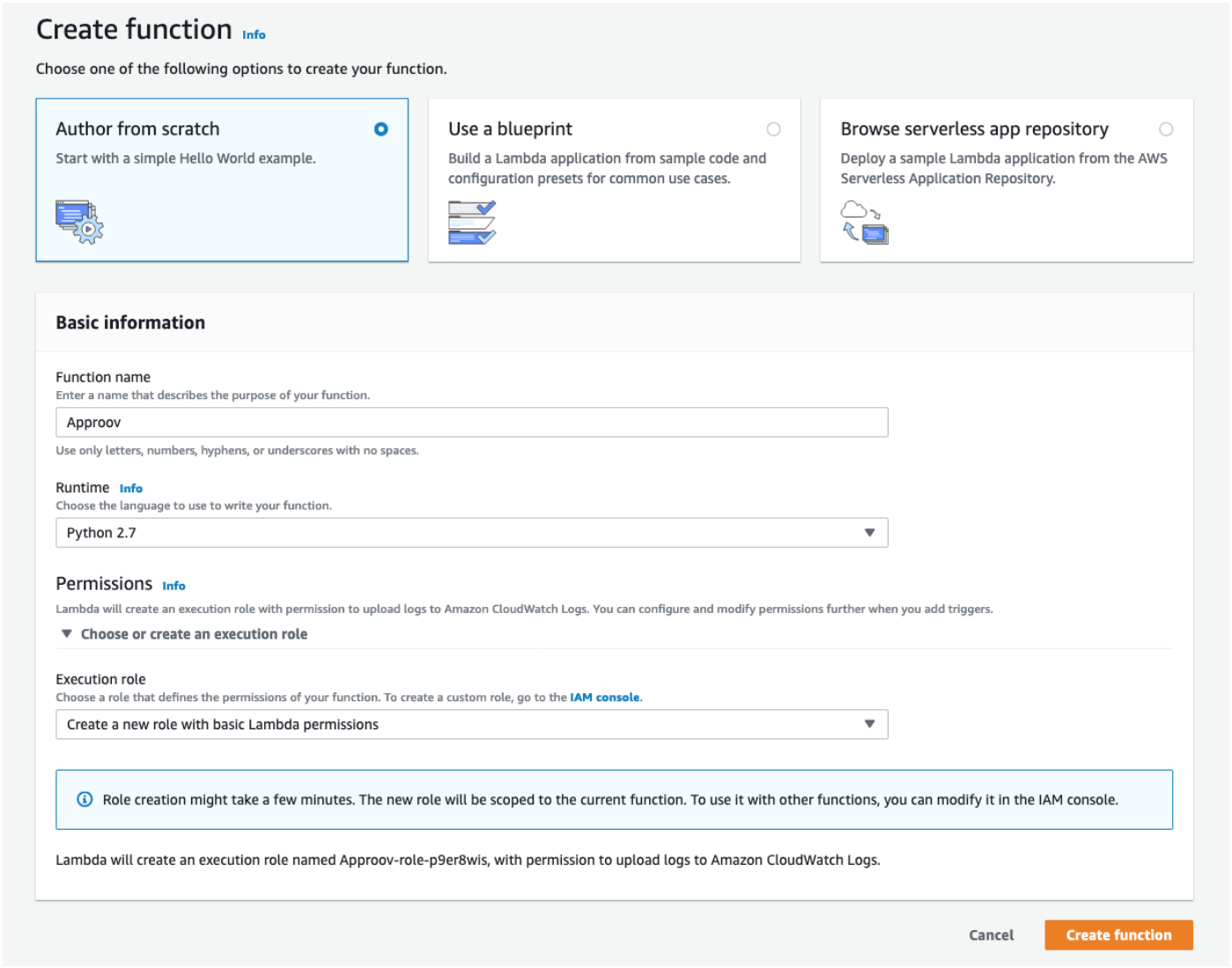
Our function is called Approov.

Go to Lambda/Functions:



Click on Create.

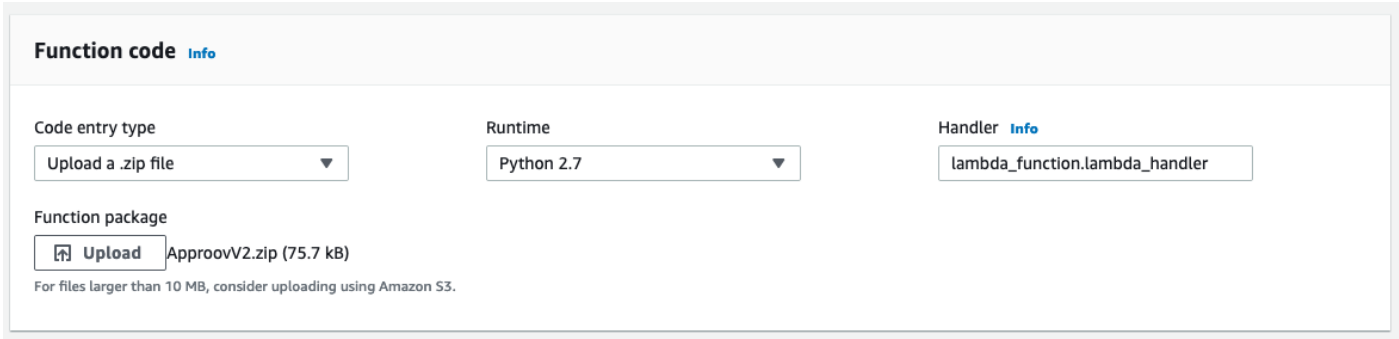
Name the function, our runtime is python 3.6, and we will create a new basic role for the lambda function to allow our API gateway stage to execute the lambda function when required.



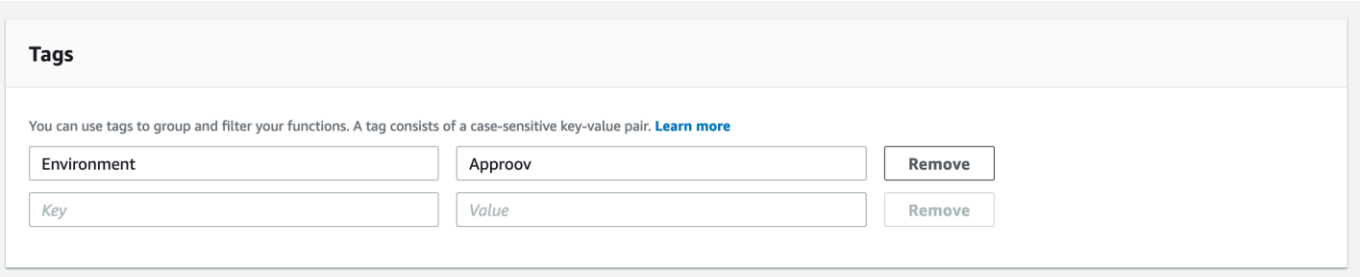
Click create.

Change function Code to “Upload a zip file”, the runtime is Python 2.6, we will keep our Lambda Handler as lambda_function.lambda_handler (default setting).

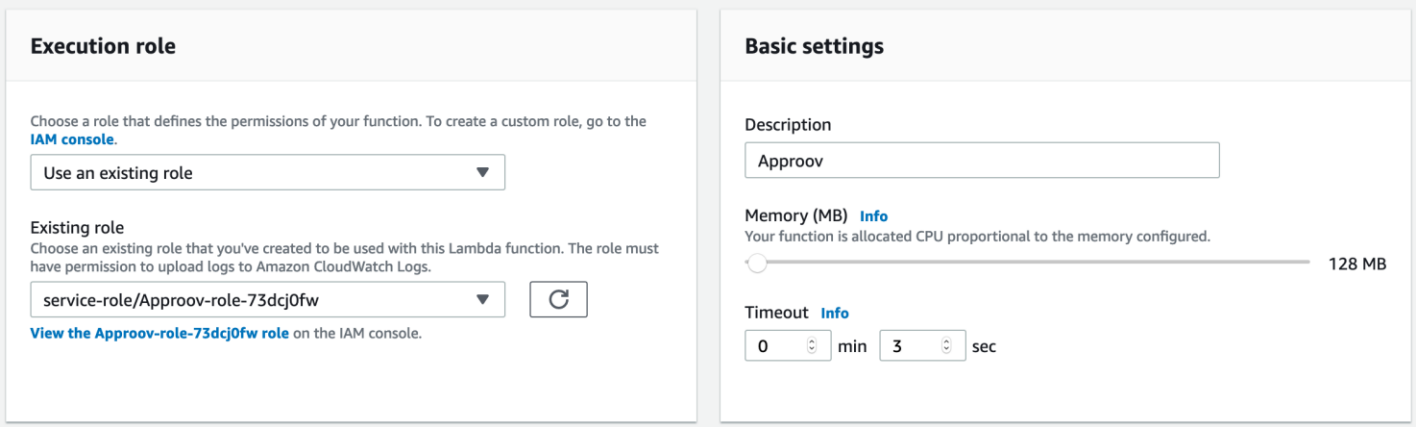
Click on upload, and upload the lambda package ApproovV2.zip



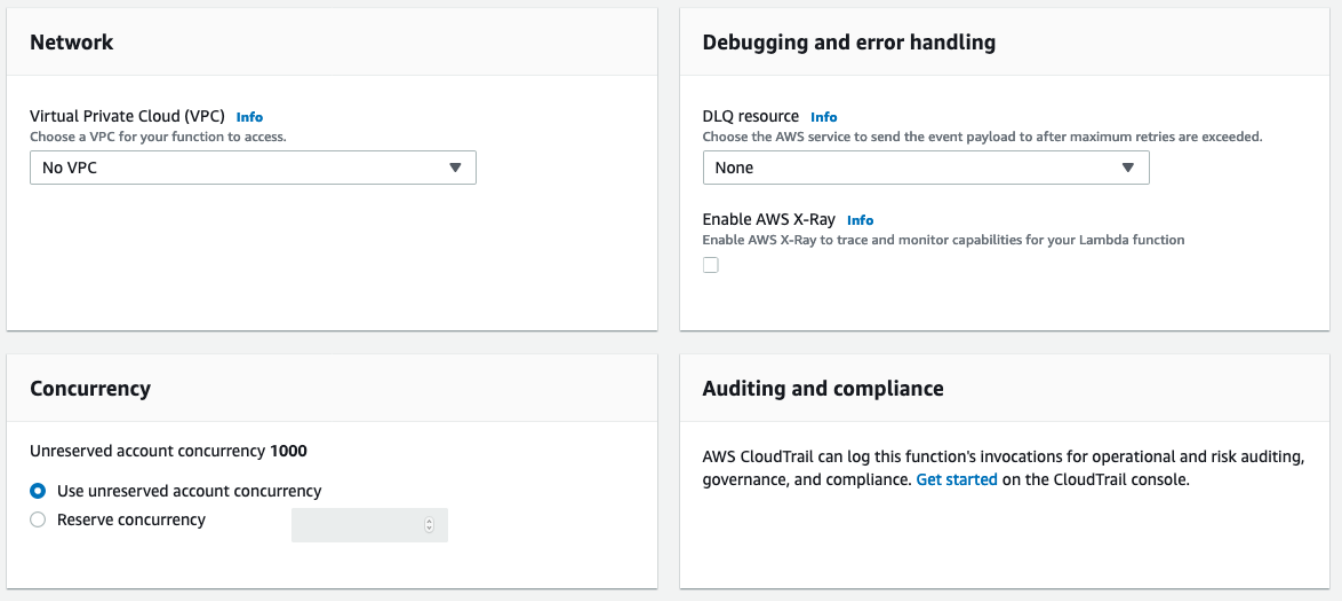
At this stage we will not set any environment variables, or encryption settings, but we will tag our function with “Environment”, and set the tag value to “Approov”.



We will use our existing basic Approov lambda role we created with the function, in my case it is service-role/Approov-role-73dcj0fw, we only need to use the default the basic setting which set the resources (and therefore charges) that our function will use. 128MB of memory, and a maximum runtime of 3 seconds after which our function will be terminated. (We cover expected resource use and execution time duration by our function, inside the cost comparisons section).



Our lambda function does not need to connect to any VPC accessible resources, so leave the Network setting with “No VPC”. For our example we will not use SQS or SNS resources for errors or debugging data.

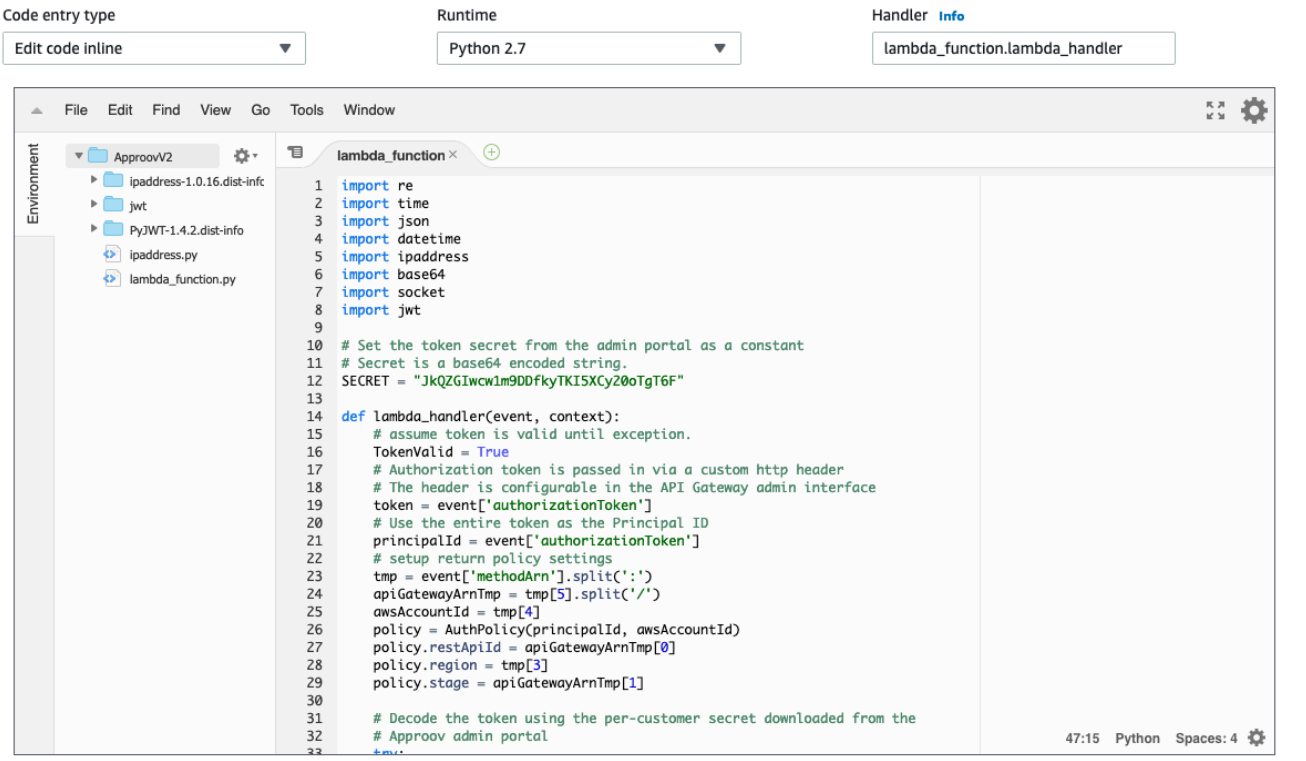


Next, Leave Concurrency as default - 1000, this is the maximum number of concurrent executions for this lambda function, and also is the maximum for all APIs that will use this function as an authoriser.

Auditing and compliance, is by default, none, and we will not activate cloudtrail.

Click the orange save button at the top right of the screen.

The function will now be displayed in the function code window.



The Function now takes the token from the HEADER AuthApproov, and decodes it (validates the token).

```
tokenContents = jwt.decode(token, SECRET, algorithms=['HS256'])
```

The jwt.decode will throw an exception for : having an invalid signature, ExpiredSignatureError (token has expired), DecodeError (unable to decode token, ie: not base64), as well as other structural errors.

We won't throw a nasty python exception and exit the function, we will set TokenValid to "False".

If TokenValid is False, at the end of processing, then the function will create a "Deny" policy and return this to API Gateway which will then return a 401 to the client.

A token will be invalided if:

- Token Signature incorrect
- Token cannot be decoded.
- exp (time expiry) claim in the token payload is non existent

If TokenValid is still set to "True" we continue process the token.

Now we need to specify the access policy to a specific API stage set within the event ARN.

```
tmp = event['methodArn'].split(':')
apiGatewayArnTmp = tmp[5].split('/')
awsAccountId = tmp[4]

policy = AuthPolicy(principalId, awsAccountId)
policy.restApiId = apiGatewayArnTmp[0]
policy.region = tmp[3]
policy.stage = apiGatewayArnTmp[1]
```

AccountID is set by the account id extracted from the event ARN (AWS Resource Name)

The PrincipalId is set to the token itself (this value can be any string that can be used as an identifier).

The RestAPIId is set the ARN ApiId of the actual API.

The AWS Region and API stage are set to the values within the event ARN.

Since we are using our API as a proxy, we will set the conditional policy to allow all HTTP methods, get,post,put,patch,delete and head.

This means that the lambda function can only be used to validate requests, and create an access policy for access to just a specific AWS account, API and stage.

If TokenValid is true, then we then create the allow access condition policy in JSON format.

```
conditions = {
    "DateLessThanEquals": {
      "aws:CurrentTime": expirationTime
    }
  }
```

The access policy is then used to set the policy conditions as well as access to all API resources in our stage, and all HTTP methods are allowed.

```
# Allow all methods, restricted to those which match condition
policy.allowMethodWithConditions(HttpVerb.ALL, '*', conditions)
```

If TokenValid is set to False, then we return a “Deny” policy with no conditions.

```
else:
  # token is invalid, deny access when token presented.
  # Deny all methods.
  policy.denyMethod(HttpVerb.ALL, '*')
```

We then build the policy, using all the settings in the policy class instance into an IAM JSON structured policy and return it to the caller (API gateway).

```
'"finally, build the policy and exit the function using return"'
return policy.build()
```

If the current time is less than the timestamp set in the token payload then the requestor will be returned the data and a 200 HTTP status.

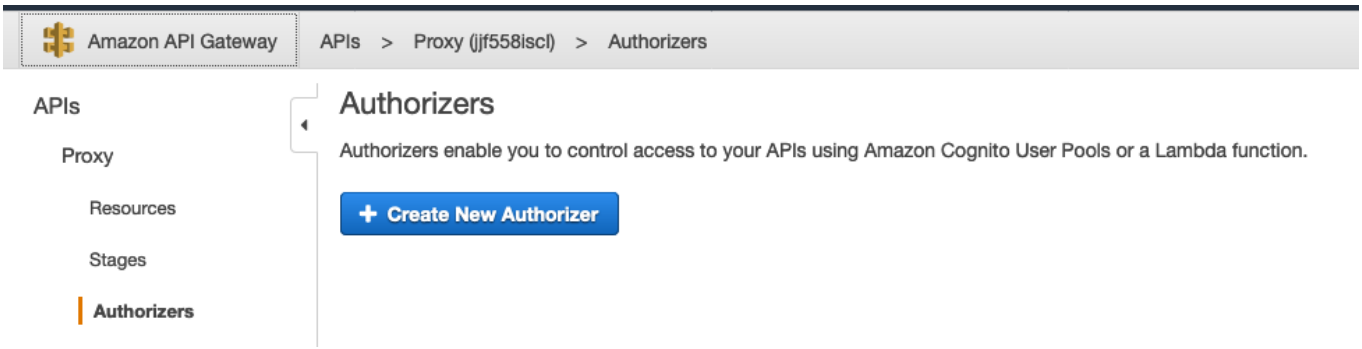
NOTE: *The access policy is used by API Gateway to give (or deny) access to APIs, stages, and endpoints, with conditions attached, namely that the current time, has to be equal or less than the timestamp held inside the “exp” client inside the token.*

This policy is then generated and returned to API Gateway to evaluate, the requesting client is then given access to the data, if the request matches the access permissions and conditions.

Here is an example policy, allowing access to endpoints of an API with an id of kjf858iscl, region us-west-1 and AWS account 645529279822, and it allows access to invoke all endpoints within all stages of this API.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Principal": "*",
         "Action": "execute-api:Invoke",
         "Resource": [
            "arn:aws:execute-api:us-east-
1:645529279822:kjf858iscl/*"
         ],
         "Condition" : {
            "DateLessThanEquals" : {
               "aws:CurrentTime": "2019-03-21T15:45:34Z"
            }
         }
      }
   ]
}
```

Now go to API Gateway, Proxy API, Authorisers.



Click on Create.

We will call the authoriser “Approov”, and we will use the lambda function Approov, that we have just created within the us-east-1 region, Choose type Lambda.

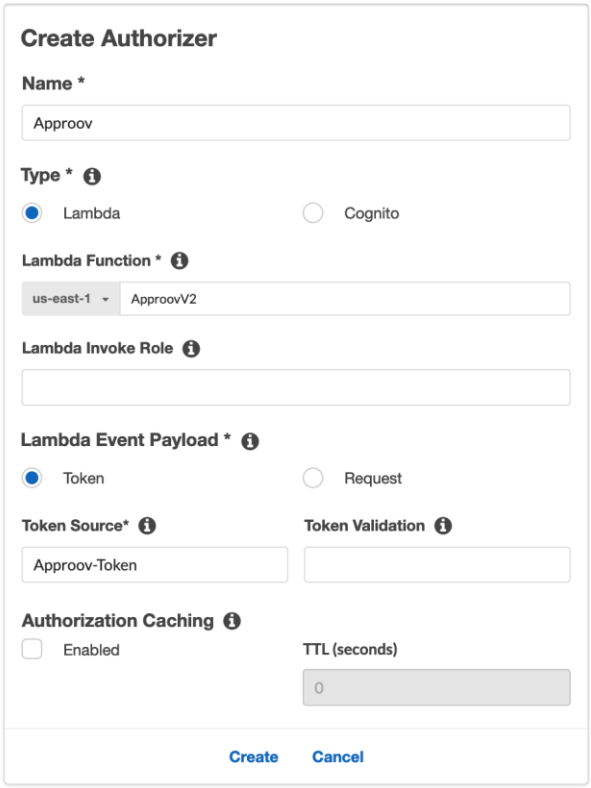
We also need to state the Token source, we will use “Approov-Token” as our Header.

Choose Token, for “Lambda Event Payload”.

We don't need to set the Lambda invocation Role here, as the role will automatically be created and granted to the authoriser, when the authoriser is itself, created.

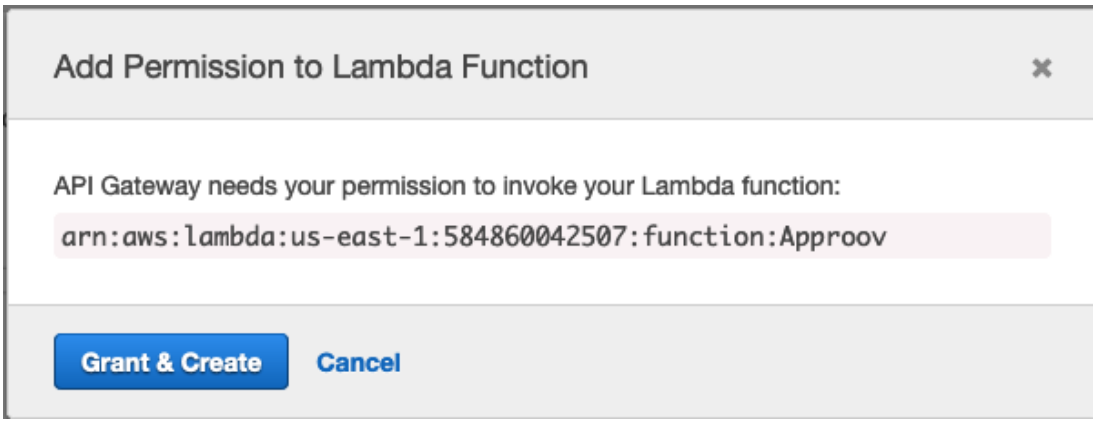
NOTE: *We won't create a token validation, just yet. I will try to keep this simple.*

We will however, enable the authorisation cache, that is a cache that stores the token as a cache key with the JSON access policy as the value, this just allows us to process the token validation once and not constantly validate and create the conditions for a set token, over and over again, so that when the token is presented the API will look within the authorisation cache for the token and if it exists loads the policy and evaluate the conditions to allow or disallow access to the requested data and HTTP method.

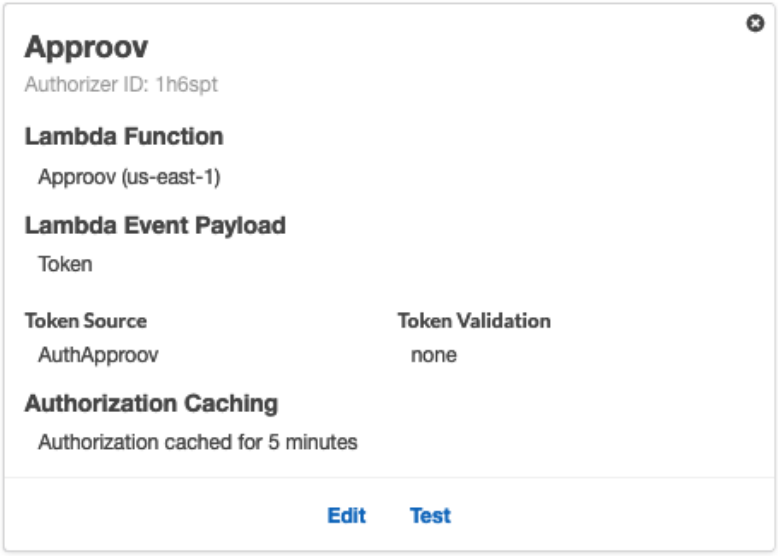


Click Create/Save.

The following window will appear...



Click Grant and Create, we now have the following authoriser configured with our API stage.



Now we need to create a long-dated token using the generator.py, to test our configuration.

You will need to change the secret inside generator.py to your secret inside the lambda function.

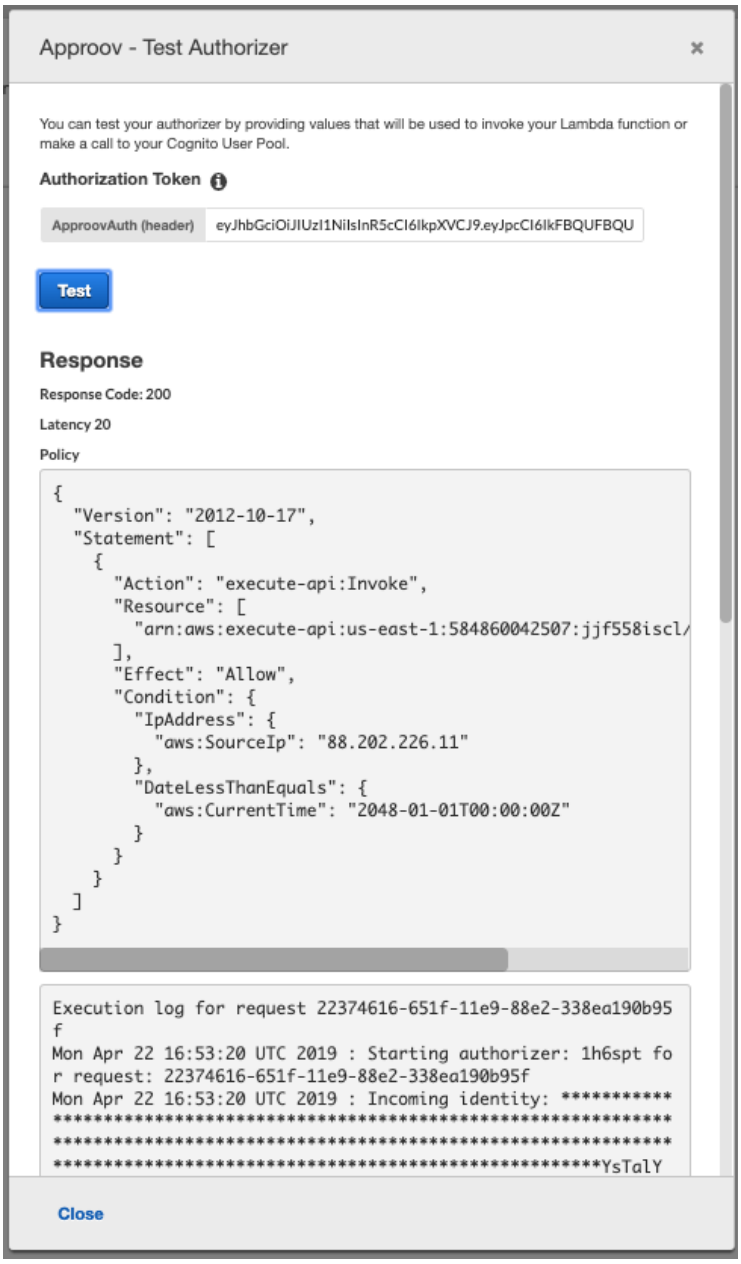
NOTE: *The token is made up of the JWT structure <header>.<payload>.<signature>*

the signature is a hash of <header>.<payload>, the validator will create a hash of the <header>.<payload> and compare the generated signature to <signature>, if it is a match, then the token is valid and the authoriser will then create the conditions and the access policy.

Please refer to Generating Long Dated Tokens section for details on how to create the token.

Now Click on “Test”.

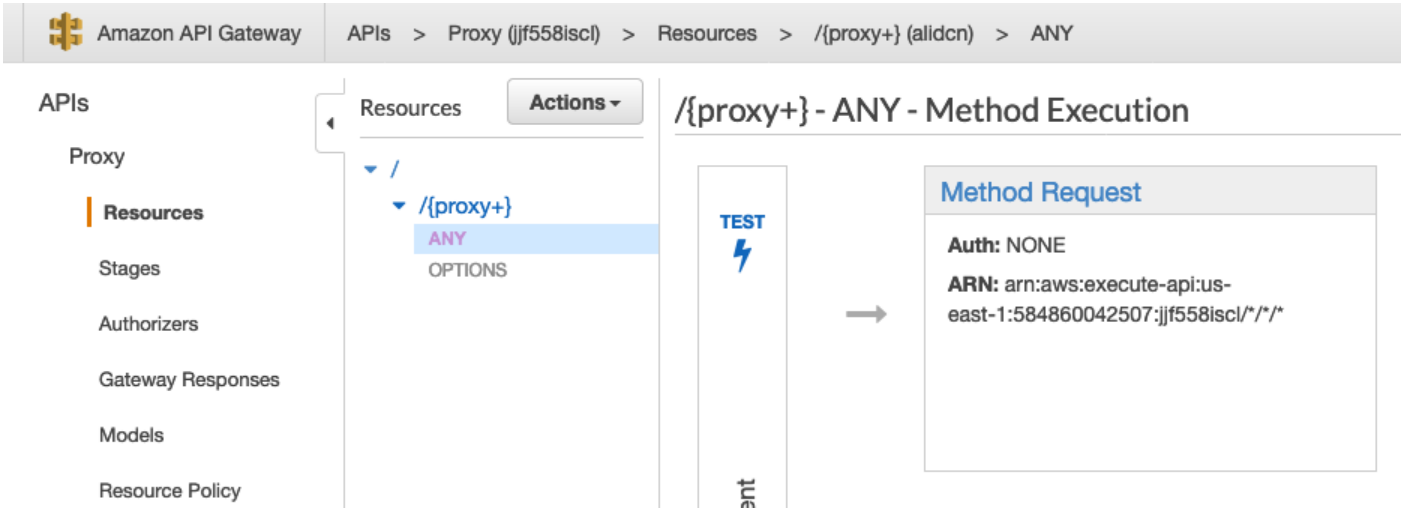
Copy the token value into the AuthApproov header:



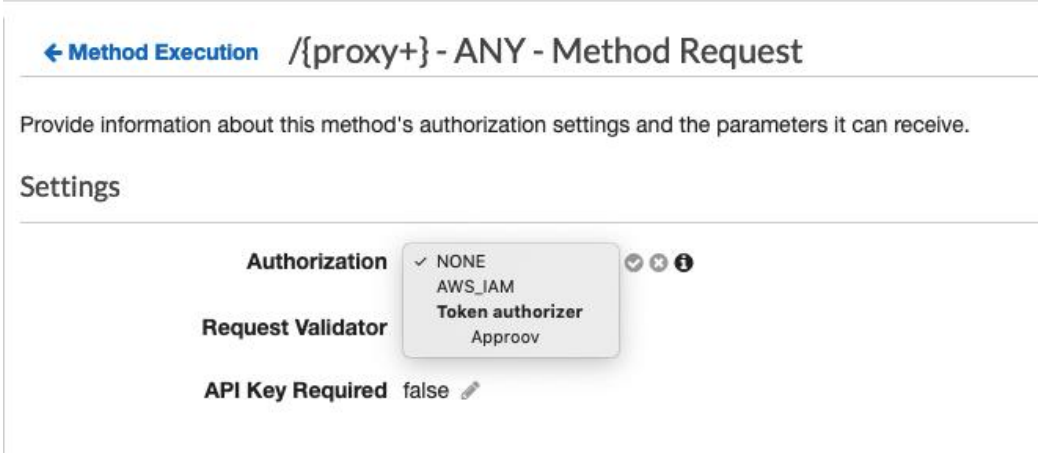
Click Test, this will return a 200 code and millisecond latency amount (my example: Latency 20) for the execution time.

This number will be higher than normal (upto x8 higher) first time the function is executed, as the function will need to cold start if it has not been executed before, click on test a few more times and this latency number should return sub 40 millisecond execution times.

We now need to attach the authoriser to the API resource, in our “proxy” API.

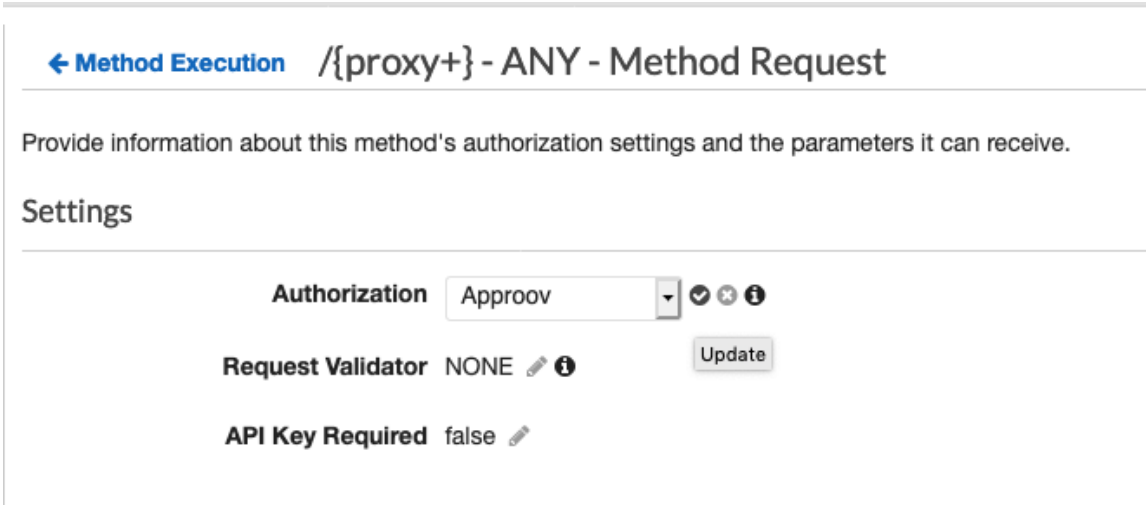


Click on Proxy/Resources, ANY, Method Request.



Click Authorisation, Approov.

Then click on the tick at the right handside, to save the setting.



Now use curl to test the token access:

```
 #
 # export token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpcCI6IkFBQUFBQUFBQUFBQUFQLy9XTXJpREE9PSIsInVpIjoiU
HJveHlUZXN0IiwiYWQiOiJpbnRlcm5hbCIsImV4cCI6MjQ2MTQ0OTYwMH0.UO3a56pIc4FPMiQfu5WVuh8MwT2QHa71hcEG33eo2Y0
 # curl -H $token https://api.dataproxy.io/example.json
[{"menu": {                                                                                                  ]
    "header": "SVG Viewer",
    "items": [
        {"id": "Open"},
        {"id": "OpenNew", "label": "Open New"},
        null,
        {"id": "ZoomIn", "label": "Zoom In"},
        {"id": "ZoomOut", "label": "Zoom Out"},
        {"id": "OriginalView", "label": "Original View"},
        null,
        {"id": "Quality"},
        {"id": "Pause"},
        {"id": "Mute"},
        null,
        {"id": "Find", "label": "Find..."},
        {"id": "FindAgain", "label": "Find Again"},
        {"id": "Copy"},
        {"id": "CopyAgain", "label": "Copy Again"},
        {"id": "CopySVG", "label": "Copy SVG"},
        {"id": "ViewSVG", "label": "View SVG"},
        {"id": "ViewSource", "label": "View Source"},
        {"id": "SaveAs", "label": "Save As"},
        null,
        {"id": "Help"},
        {"id": "About", "label": "About Adobe CVG Viewer..."}
    ]
}}
 #
```

Once the Proxy API, using the authoriser and client certificate(or apikey) to access the origin, and has successfully returned our example data, that resides on the origin, we can then allow API gateway to cache both the backend data, and the token and conditional access policy.

CACHE

WHY AND WHEN

API Gateway has the ability to cache data returned from the origin, the data is cached by its full URL, including path parameters and query strings.

API gateway also has the ability to cache the access policy of the token after the token has been validated by the authoriser.

These two caches can be enabled and configured, independently of each other.

There are benefits and disadvantages to enabling caching, especially the data cache.

Advantages for enabling the data cache:

- less requests are sent to the origin, and the data is cached closer to the client.
- less load on the origin, means potentially, less resources at the origin, which means less cost.

- requests are processed faster, if the data is in the cache the data will be immediately returned, increasing overall performance of the app when accessing API data.

Advantages for enabling the token cache:

- lowered cost, as the authoriser is not executed every single time the same token is presented in every request, it is validated just once and cached with the conditional access policy.
- performance: once the token has been evaluated and the access conditions cached, API gateway will access this policy for the request before it executes the authoriser. Inside a high traffic environment this will generally shave about 30 to 40 milliseconds off every request.

Disadvantages to enabling data cache:

- TTL (Time-To-Live) of the data needs to be set appropriately, the higher the TTL is set the longer it will take for new data to appear in the API, assuming cache entries are not manually invalidated.
- Setting a high TTL will decrease the load on your origin, you might however have to design and implement a cache management system to override the TTL value for rapidly changing data. This basically means having a process that can refresh individual cache entries when required, increasing cost and overall system complexity.
- Logging required for monitoring purposes (like troubleshooting outages), it is also required if you want to use this data for performance monitoring and big data analytics on requests and requestor data.

Disadvantages of enabling Token cache:

- Unlike the data cache, token cache there is no method to invalidate entries, unless the entire cache is flushed.
- Cache should be set to a maximum TTL to what the amount of time the token’s time expiry is set to.

The simplest, overall, solution might be to disable the data cache, but keeping the token cache.

Advantages to disabling data cache:

- Access logging not required at API, most if not all metadata regarding requests are forwarded to the origin where the origin can store the metadata in local access logs.
- No data cache management required.
- All modified data is immediately available, there is no wait for the old value to expire inside the cache
- Simplicity, no overall management, it's just a reverse proxy, that secures your data to only your Approov protected App.

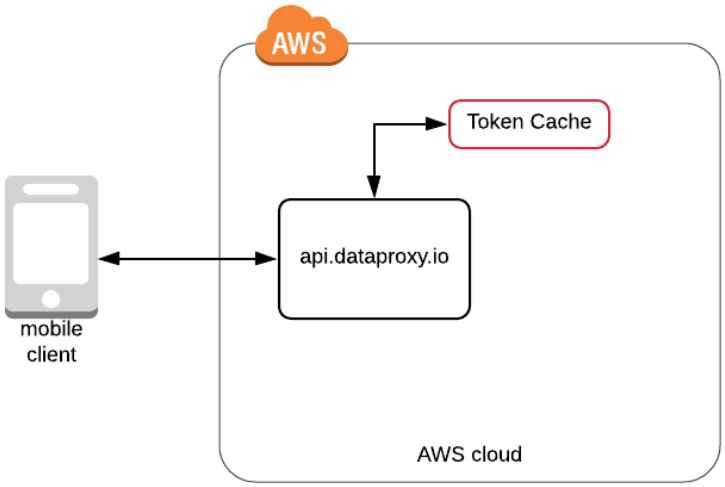
Disadvantages to disabling the data cache:

- Possible traffic spikes, all valid requests will go through to the origin.
- Origin must have resources available to handle any traffic spikes.
- Possible increased cost for additional resources at Origin.
- Increased data cost for traffic travelling from (out) API Gateway to the origin.

NOTE: *Cache time-to-live (TTL), sets how long the cache entries are cached for before being automatically invalidated and refreshed from the origin when the next request arrives, this value can be set from 60 seconds to 3600 (1 hour)*

The data cache can differentiate between data from different query string values - however caching of data for specific Query string values, needs to be set inside the API configuration for method request, this is explained in this section.

TOKEN CACHE



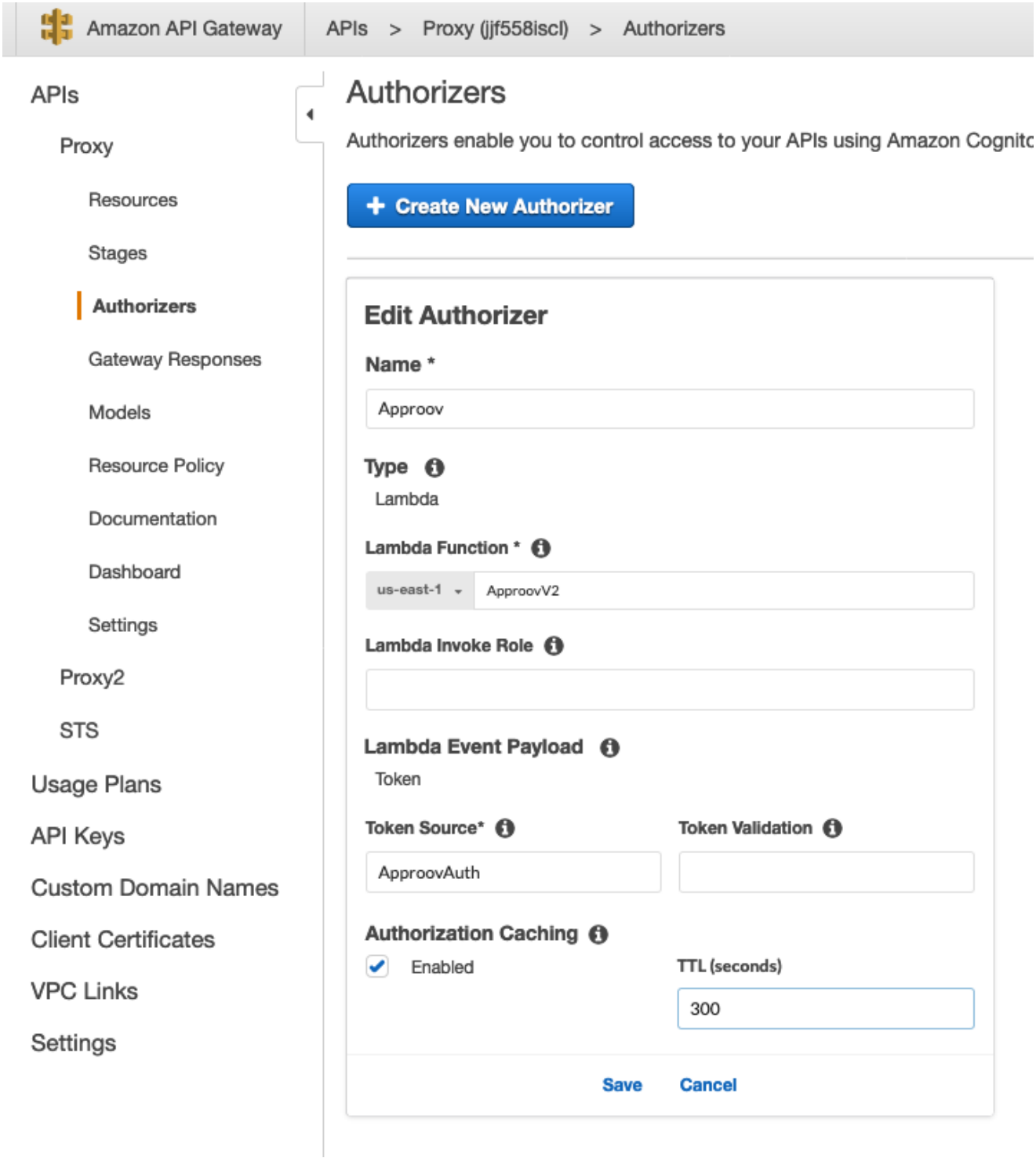
The Authorisation (token) cache is used, primarily for validation of tokens only once, then the token and conditional policy is cached allowing further requests to be validated, without the overhead of validating the token via a lambda execution. Authorisation cache lowers cost, and increases performance.

The token cache operates in a similar way to a key-cache database. With each request, API Gateway will check to see if the token is stored, and if so, gets the access policy and evaluates the request against the policy and its conditions, so it can either allow or deny access to the data via the API specific endpoint.

If the token cache is enabled tokens can be cached for between 1 to 60 minutes.

TIP: *only set the Token Cache Time-to-live to what interval you have set inside the Approov Administration console. If your tokens are set to expire every five minutes, set the Token Cache TTL to be 300 seconds (5 minutes) or less.*

To enable the token cache on your authoriser, click on the Approov authoriser and enable the cache, the default is 300 seconds, which is 5 minutes, for our example.



Policy Conditionals

These are included in the policy associated with the token.

These are just checks, the only one we use is to check that the token has not yet expired.current date time at must be less than the expiry time set inside the token payload, in this case 2019-03-21T15:45:34, for the request to be allowed access to what is set within the policy.

Example:

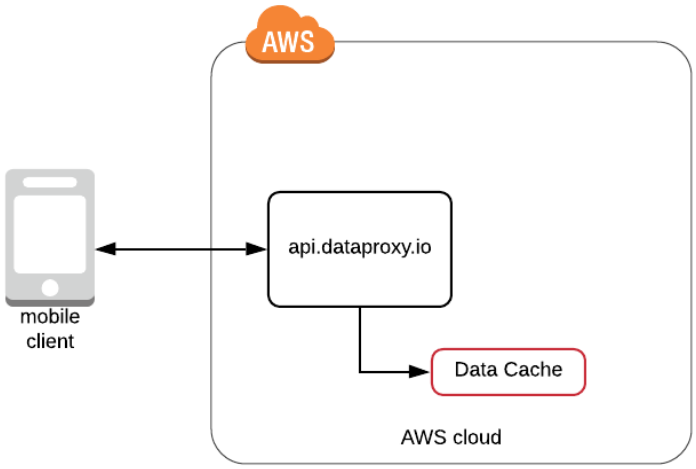
```
"Condition" : {
        "DateLessThanEquals" : {
            "aws:CurrentTime": "2019-03-21T15:45:34Z"
        }
```

Invalidations

Once a token policy has been written to the token cache, there is no existing method to invalidate a specific token. The token will however be flushed from the cache when the time-to-live has expired.

DATA CACHE

If the data cache is enabled, data (only GET) content can be cached between 1 to 60 minutes.



Note: Make sure the Method Request and Method Integration sections in resources of the API, before it is deployed, are set:

Within Method Request, enable the cache :

Within Method Integration:

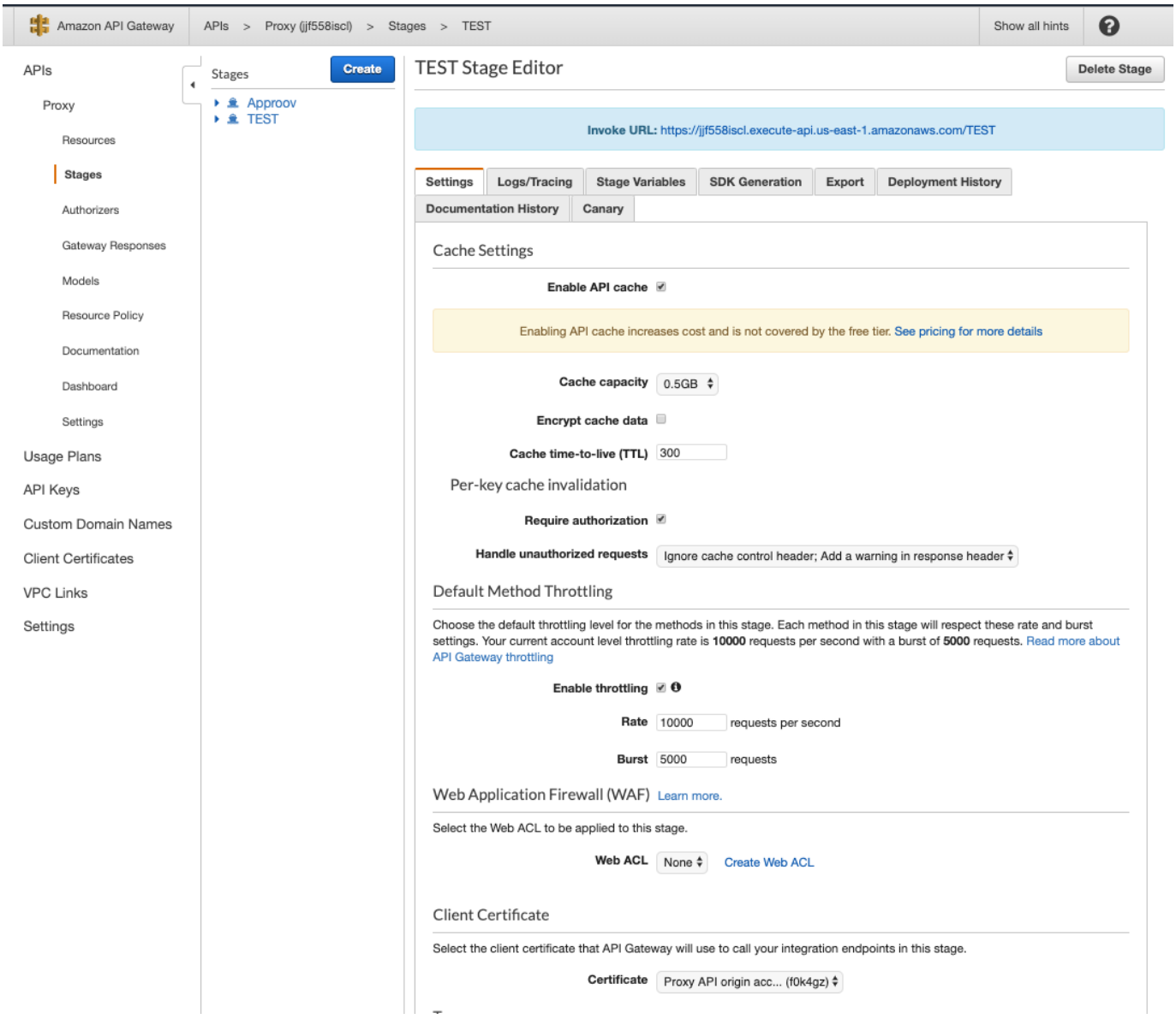
You can now set the data cache inside the deployed API stage:

We will enable the data cache.

Settings:

The data cache is set on the stage (our custom domain is mapped to our API “TEST” stage)

To set the cache, click on the stage:



We will set our API data cache capacity to only 0.5GB, we will not enable cache encryption, we will set our TTL to the default 300 seconds/5 minutes, and we need to require authorisation for individual key invalidations (explained below).

Keep request throttling at the normal 10000 per second, this is actually quite high for a small usage API, but since we are protected by Approov, we can leave this on default.

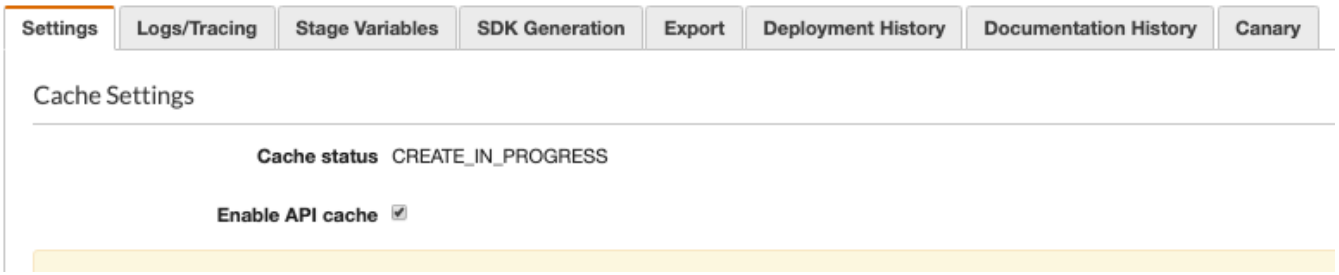
Web ACL of the AWS WAF, we will not use, please refer to the end of this section on why the WAF is not enabled for our API.

Make sure the client certificate is set to the one installed on our origin.

Then click on Save.

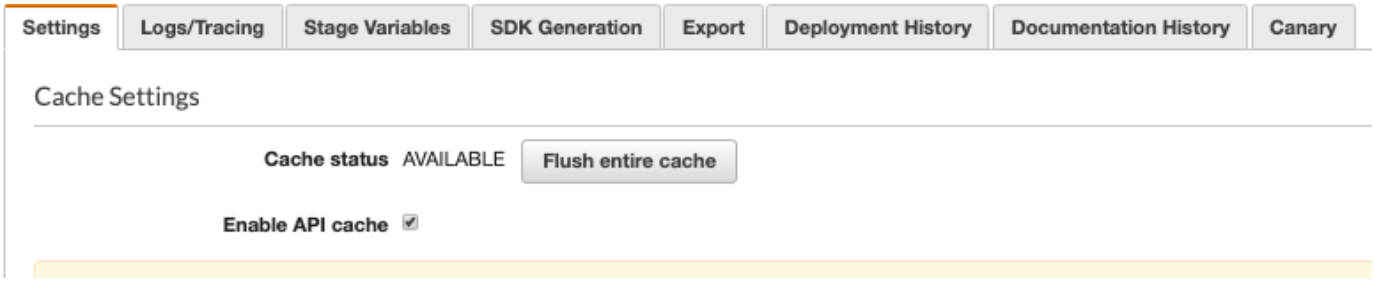
This will create the data cache, it will take a few minutes to be fully activated.

This will show the create in progress status.



Data Cache Flush and invalidations:

To flush or empty the entire data cache, inside the console, click the Flush entire cache button.



This should really only be executed if the entire cache is filled with incorrect data or the cache appears corrupted, the cache should only be flushed in a maintenance window, as flushing the cache can potentially overwhelm the origin with requests until the cache is filled again.

The specific data cache entries can be invalidated (refreshed) individually through an authorised *cache-control:max-age=0* request header sent with the URL request into our API Proxy, this allows the request to avoid the cache data and retrieve the data from the origin itself, an additional feature is that the returned data (if the return is a 200) is stored in the data cache.

This can be completed simply by a lambda function or EC2 instance that has a role with an IAM policy that grants the execute-api:InvalidateCache is authorised to invalidate cache entries.

An example policy to allow for authorised invalidations, would be:

This would allow the role user to invalidate any endpoint data held in the data cache for stage TEST.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "execute-api:InvalidateCache"
      ],
      "Resource": [
        "arn:aws:execute-api:us-east-1:645529279812:kjf858iscl/TEST/GET/*"
      ]
    }
  ]
}
```

Always set “Require Authorisation” on the data cache.

WARNING: *if “Require Authorisation” is set to off, ANY request using a header:value of cache-control:max-age=0 will be able to invalide ALL data points inside your cache and potentially overwhelm your origin with requests.*

AWS WAF

Both AWS WAF and Approov are categorised as Intrusion Protection Systems (IPS).

These systems are designed and built to prevent malicious and unauthorised requests from gaining access to the systems that they control access to.

AWS WAF can either be created with your own rules or it can use one of the many useful WAF configurations from the MarketPlace (like F5).

The AWS WAF sits between API Gateway and clients (Mobile Apps) in the public Internet, this means that the WAF will filter the traffic before the traffic hits the API gateway and our token authoriser.

The authoriser will always deny access to requestors that do not have a valid token.

Therefore Approov acts better than a WAF, in only allowing access to the data from clients that include a valid token with the request.

All AWS services including API gateway, are protected, for free, by AWS Shield standard.

AWS Shield Standard is configured for Network flow monitoring and mitigation from common DDoS attacks.

SERVERLESS API MANAGEMENT

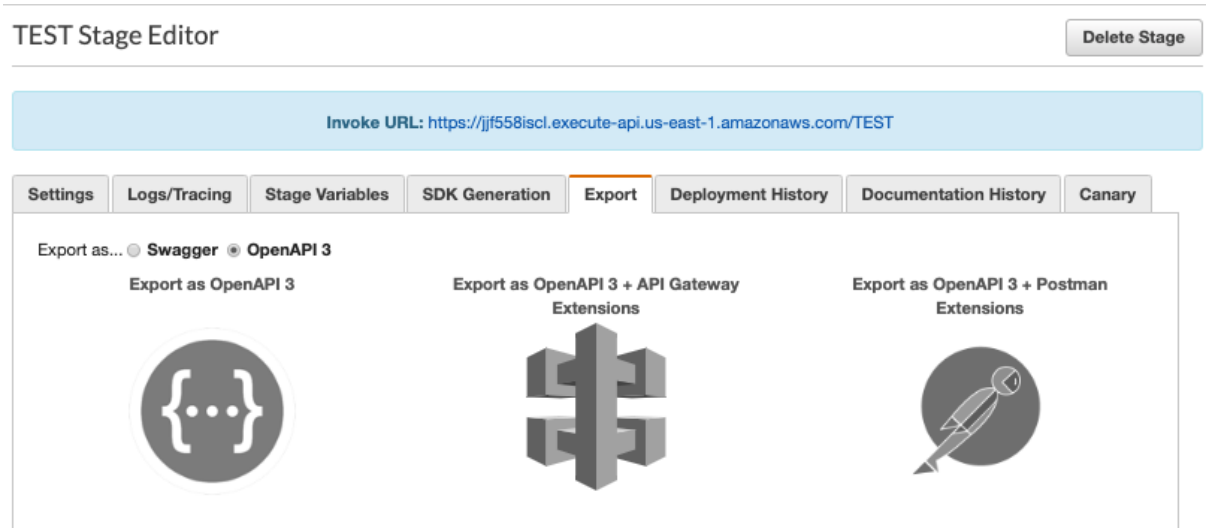
MANAGING THE API

Once the API is all setup there are various ways to manage, modify and backup the configuration and the associated lambda functions.

Swagger (openapi) is a configuration format for exporting your API configuration and or creating a or cloning a new API.

Only deployed stages can be exported.

In the console, go to TEST stage, and select the “Export” tag.



Select to export as OpenAPI 3.

Then “Export as OpenAPI 3 + API Gateway extensions.

you can either download the export in YAML format or JSON.

My example is JSON.

This is an export of the API configuration.

Note: the export does not contain any of the following:

- Authoriser
- Approov lambda function
- Client certificate
- Custom domain

The export is used to version your API, for example, endpoints or structure, of a particular deployed API (stage).

For example, to use an export as an import to create a new API:

Open up the downed JSON export.

Change the Title from “Proxy” to “Proxy2”.

NOTE: *If you do not change this Title the import will overwrite the lastest resource configuration of the API "Proxy", the deployed stages will be left unchanged, only the latest configuration still held in resources will be overwritten.*

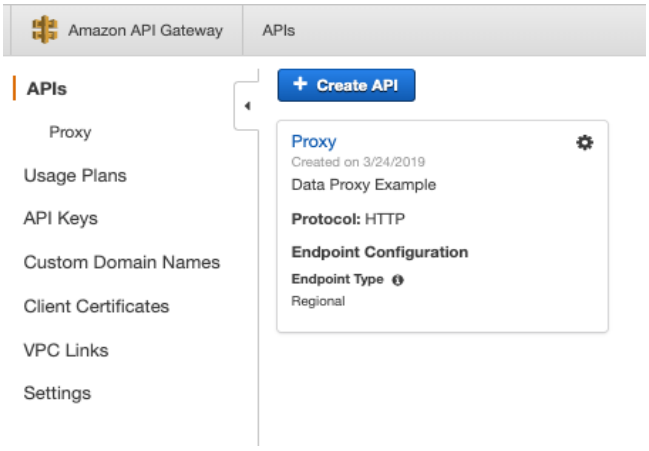
Now you can add any additional endpoints or modifications to the API structure before the upload.

If we change the title to "Proxy2" and create a new API with the export it will be identical to Proxy but it will create a new API called "Proxy2".

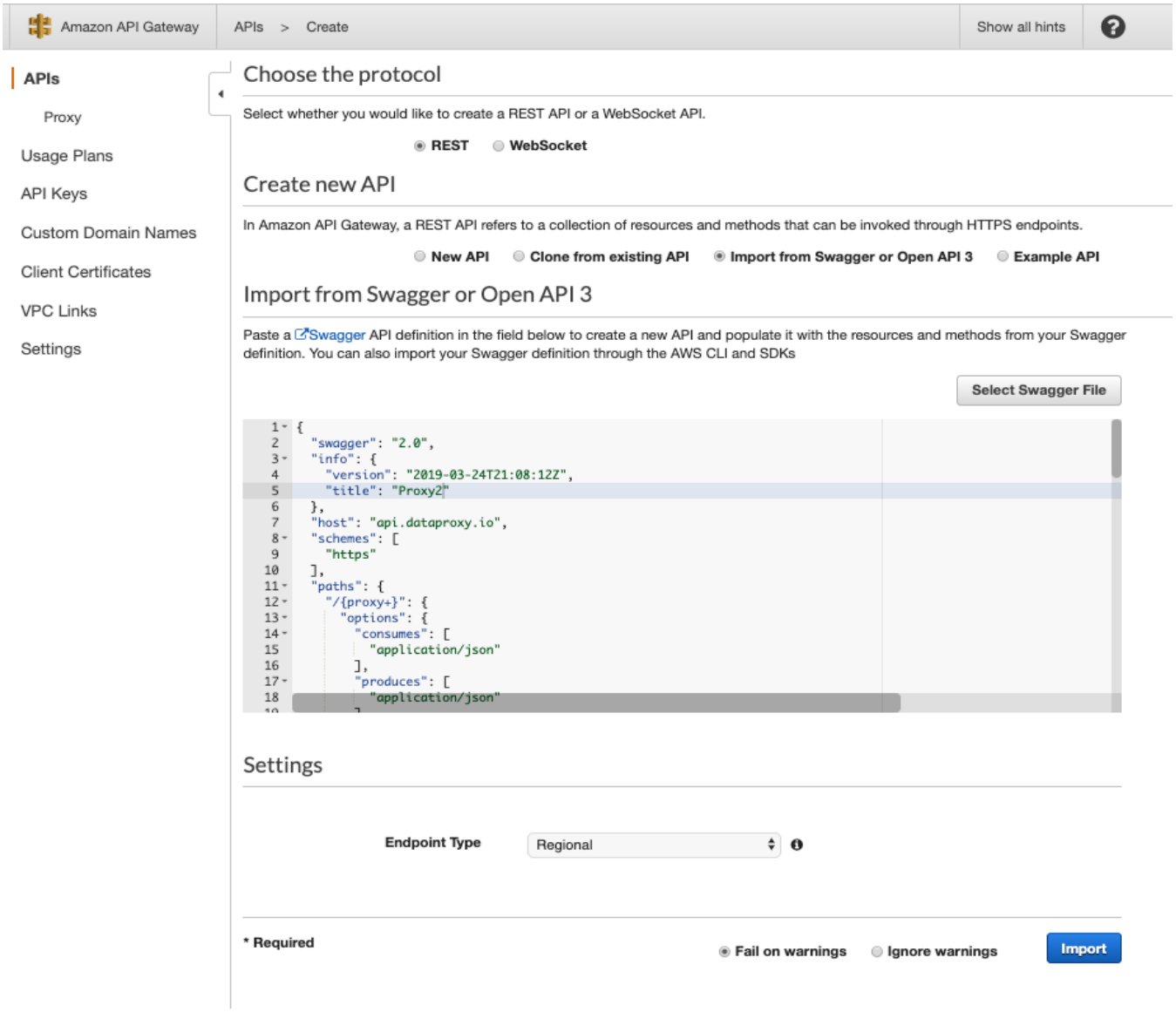
The change is as follows:

```
{
  "openapi": "3.0.1",
  "info": {
    "title": "Proxy2",
    "version": "2019-03-24T21:08:12Z"
  },
```

To create a new API, click on APIs, Create API.

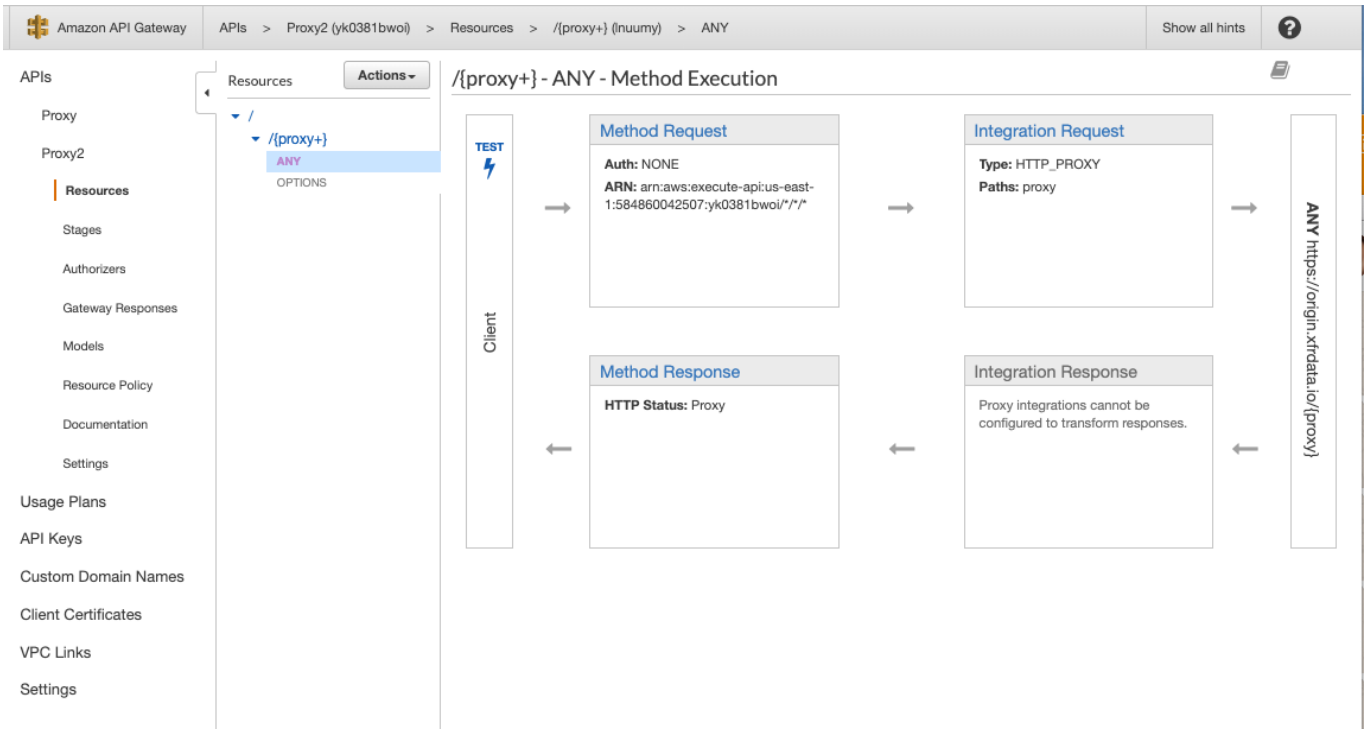


Click Rest, Import from "Swagger or Open API 3", select swagger file, as in the file you just downloaded and changed the Title to "Proxy2", keep Endpoint type regional.



Click import.

Now you have a new proxy2 API.



If you notice there is no Auth(authoriser) within Method Request, there is no stage deployed (so no client certificate attached).

For building out a full API, inclusive of functions, certificates, route53 domains and integrated configurations (so that it all works from the first time build), you will need to use toolsets like Cloudformation, Terraform. Ansible and some other continuous deployment systems will also be advantageous to update environmental settings, and lambda function code.

CHANGING THE SECRET

First, get the secret from Approov via the Approov tool CLI, for instructions on how to install and configure the Approov CLI Tool, go to Generating Long Dated Tokens.

Once the CLI is configured, get the secret:

```
# approov secret -get
JkQZGIwcw1m9DDfkyTKI5XCy20oTgGT6D
```

There are two methods to managing the secret for the authoriser, method1 is static, method2 is dynamic.

Method 1: Static Variable inside Authoriser

In our authoriser the secret is a static variable inside lambda function handler.

Simplest way and the most manual is to change the secret within the lambda code window inside the console, or alternatively by changing the secret inside lambda_handler.py, recreating the package and uploading this into AWS via the console or uploading via the AWS Command Line Interface (CLI).

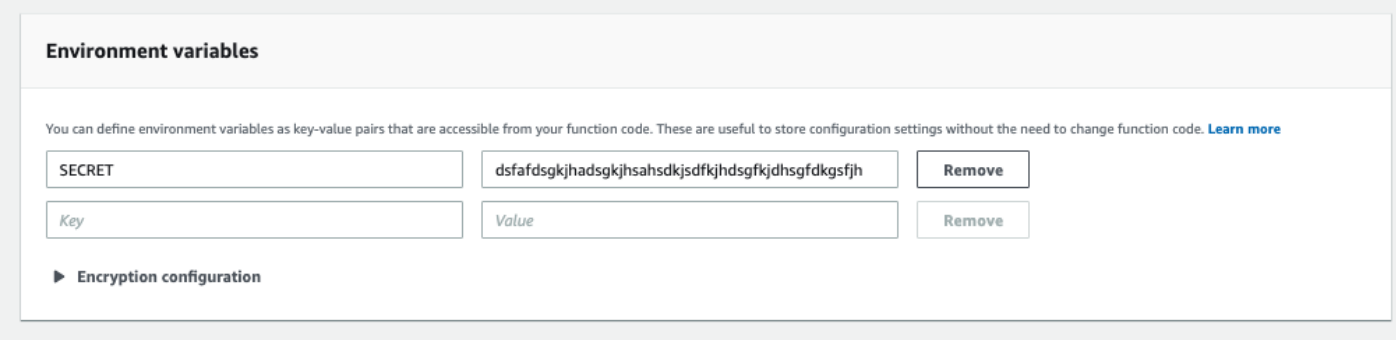
When you save a new version of the function, the lambda function will become “LambdaFunction:$LATEST” (we will not delve into function versioning).

Method 2: Environmental variables sourced from inside the lambda function

This method of storing the secret is the most flexible, you create a lambda environment variable that is associated with the lambda function but sits outside of its codebase, this means you do not create a new lambda function with the secret embedded in it, it means you can dynamically modify the secret either within the console or via the AWS CLI without touching the lambda function itself.

Example: To first create an environment variable go into functions/Approov.

Create a new Environment var like the below example:



The existing default authoriser lambda function has the secret embedded within it as a string:

```
import jwt
# Set the token secret as a constant
SECRET = "JkQZGIwcw1m9DDfkyTKI5XCy20oTgGT6D"
```

Modify this part of the function by placing “import os” at the top of the function and using os.environ to source our environment variable attached to the lambda function, named ‘SECRET’.

```
import jwt
import os
# Set the token secret from environment variable SECRET
SECRET = os.environ[‘SECRET’]
```

AWS TOOLSETS

AWS COMMAND LINE INTERFACE

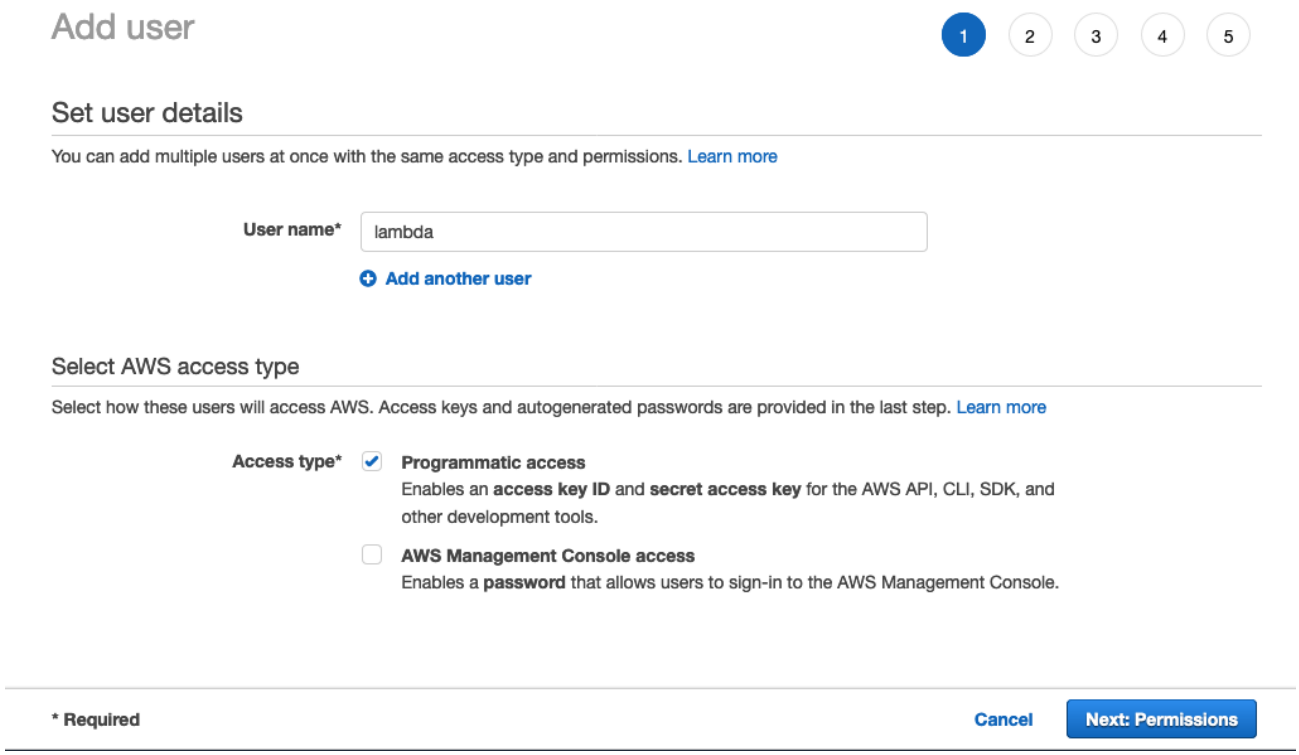
The AWS CLI is a powerful open source tool that replicates management functionality, it is equivalent in features to the AWS Console. Versions are available for both Linux and Windows based systems, via linux terminal shell or windows powershell, it does however require Python >2.6.5 to be installed.

In our example we will install the AWS CLI on a ubuntu instance, create a user with only lambda permissions, create and install a set of AWS IAM programmatic keys for use by the AWS CLI in managing lambda in AWS.

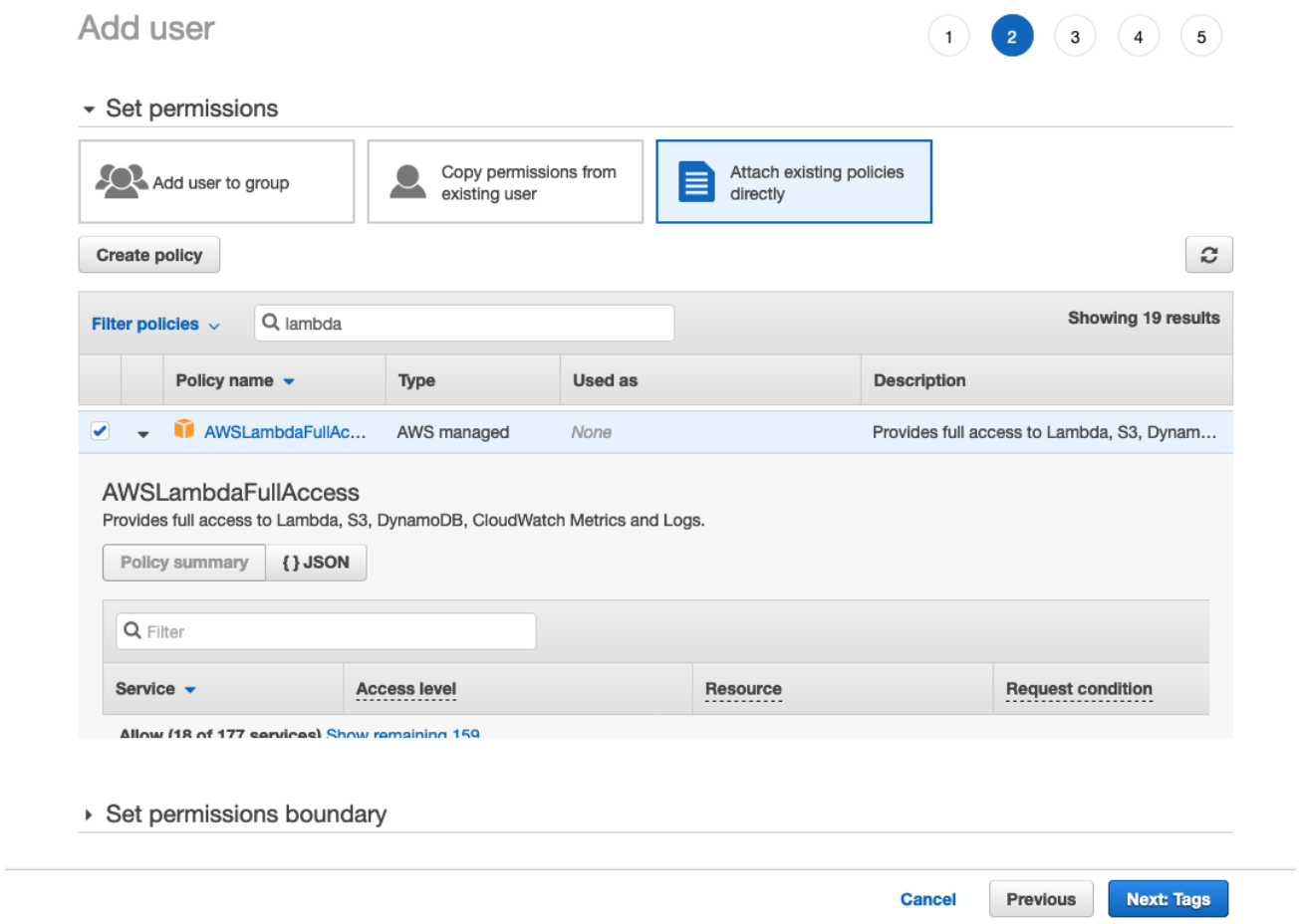
First we need to create an IAM user with the AWSLambdaFullAccess Policy.

Login to the console, go to IAM:Users

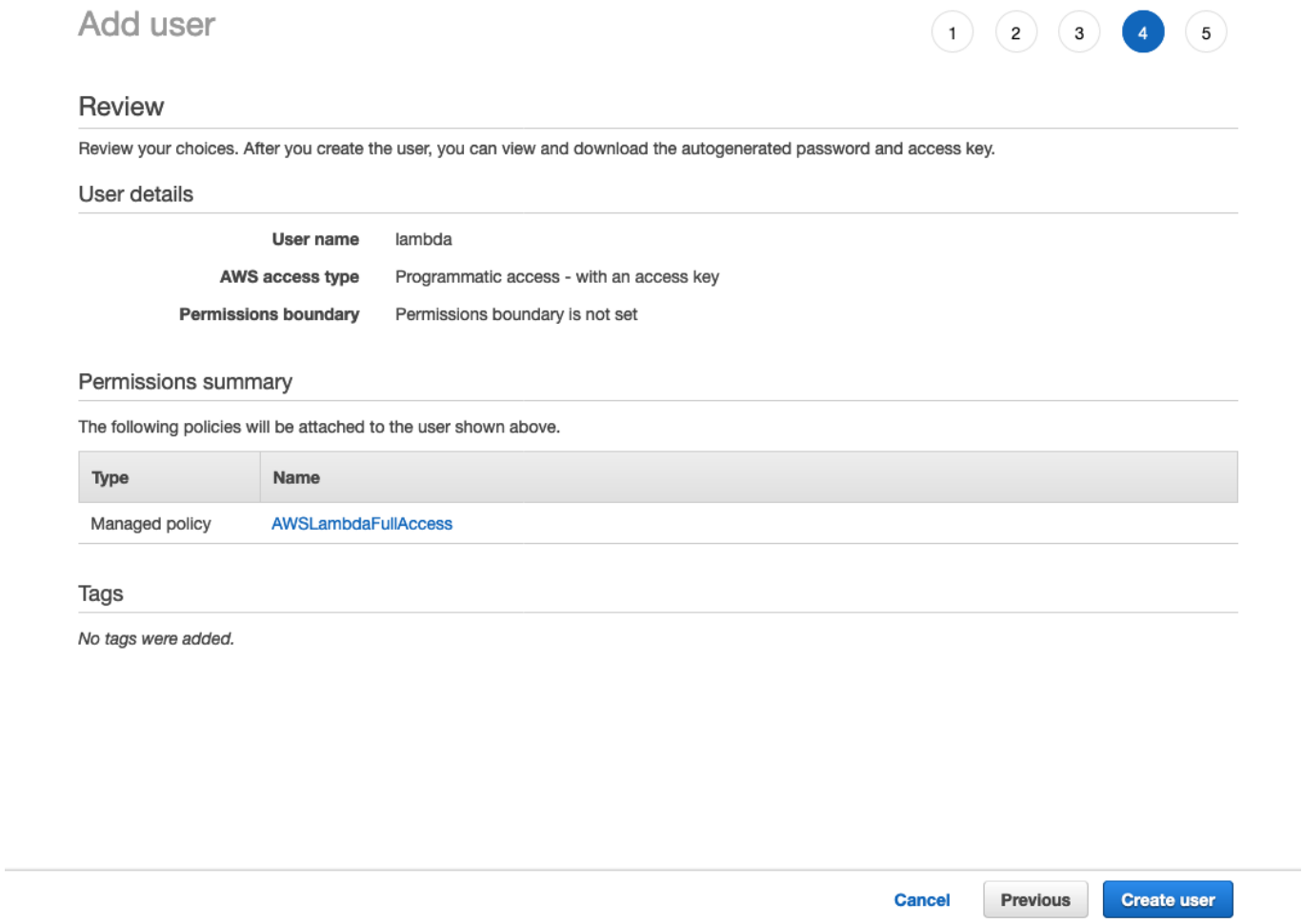
Create the user Lambda, and only enable Programmatic access.



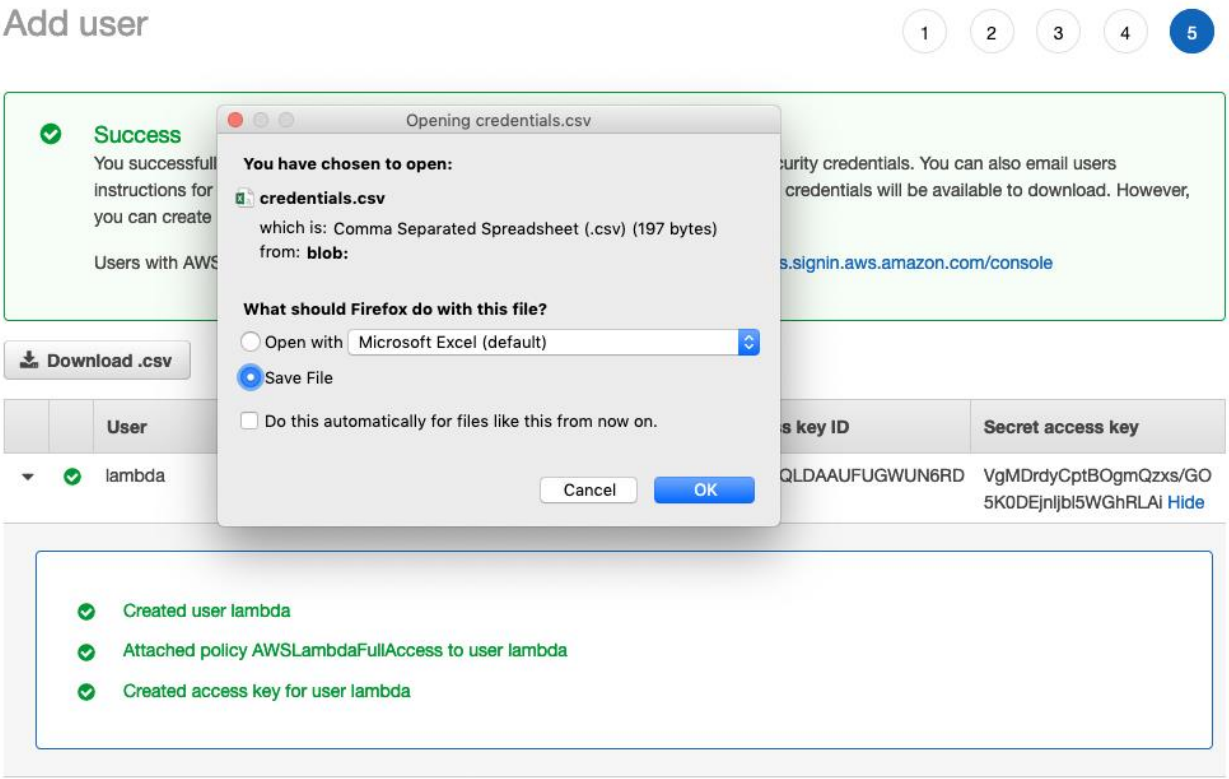
Next, attach the AWSLambdaFullAccess Policy.



Click, Next, we won't use tags in this example, click “Next Review.



Click Create User, then immediately download the csv, copy the Access key ID, then Show and copy the Secret Access key.



NOTE: ***The Secret access key is only available to you on this page, it is important that you download this information held it the csv file and store it securely. If you lose the Secret you will need to regenerate the Programmatic keys.***

You now have AWS Credentials for only managing Lambda functions within your account.

To Install the AWS CLI, you need Python version 2.6.5 or above (preferably 3.4) and pip (preferably pip3)python

In ubuntu:

```
root@host:~# python3 -V
Python 3.4.3
root@host:~# pip3 -V
pip 1.5.4 from /usr/lib/python3/dist-packages (python 3.4)
root@host:~#
```

Create the user that will manage AWS Lambda in your account:

```
# useradd lambda -p lambda --create-home -d /home/lambda -s /bin/bash -c"AWS Lambda management"
```

Now install the AWS CLI for use by the user.

```
# su - lambda
lambda@host:~$ pip3 install awscli --upgrade --user
Successfully installed awscli PyYAML colorama botocore rsa s3transfer python-dateutil urllib3 jmespath pyasn1 six
```

You have now installed the AWS CLI into user lambda's home directory (not system wide).

You will need to source the bin and lib directories within .local within your home directory.

NOTE: *I have exported these on the commandline, these two export commands should be inside your local .profile or .bash_profile.*

```
lambda@host:~$ export PATH=~/.local/bin:$PATH
lambda@host:~$ export LD_LIBRARY_PATH=~/.local/lib
lambda@host:~$ which aws
/home/lambda/.local/bin/aws
lambda@host:~$ aws --version
aws-cli/1.16.162 Python/3.4.3 Linux/3.13.0-125-generic botocore/1.12.152
lambda@host:~$
```

Now install your AWS credentials:

```
lambda@host:~$ aws configure
AWS Access Key ID [None]: AKIAYQLDAAUFUGIHN6RD
AWS Secret Access Key [None]: VgMDrdyCptNOgmQzxs/G65K0DEjnljbl5WGhRLAi
Default region name [None]: us-east-1
Default output format [None]: json
```

lambda user should now be able to use all lambda control functionality in the CLI.

To list the lambda functions within your account (there is only one function the authoriser in my example account) : type 'aws lambda list-functions'.

```
lambda@host:~$ aws lambda list-functions
{
    "Functions": [
        {
            "TracingConfig": {
```

```
            "Mode": "PassThrough"
        },
        "Version": "$LATEST",
        "CodeSha256": "qB7Af0mH6kEmZw2dKLduP2HftUtarXnJho4vYFzzAxQ=",
        "FunctionName": "ApproovV2",
        "VpcConfig": {
            "SubnetIds": [],
            "VpcId": "",
            "SecurityGroupIds": []
        },
        "MemorySize": 128,
        "RevisionId": "8ded2bf5-787f-4079-8354-87e2bf39821e",
        "CodeSize": 75143,
        "FunctionArn": "arn:aws:lambda:us-east-1:584860042507:function:Approov",
        "Handler": "lambda_function.lambda_handler",
        "Role": "arn:aws:iam::584860042507:role/service-role/Approov-role-73dcj0fw",
        "Timeout": 3,
        "LastModified": "2019-04-22T15:09:03.172+0000",
        "Runtime": "python2.7",
        "Description": "Approov"
    }
  ]
}
lambda@host:~$
```


This will also list the function's configuration, its runtime, Memory size, role, etc…

An example for managing your authoriser, is if the secret is static inside the function, you can download the function as a zip file, unzip it, change the secret inside the Lambda)_Function.py file, repackage the modified version, with the new SECRET, as a new zip file and then upload the zip file as an update to the Approov Function.

First download the file, using get-function, by default the file will be stored inside an AWS S3 bucket, referencing your account and lambda function name, download this file (reference as "Location") via a browser.


```
lambda@host:~$aws lambda get-function --function "ApproovV2"
{
   "Code": {
```

```
    "RepositoryType": "S3",
    "Location": "https://prod-04-2014-tasks.s3.amazonaws.com/snapshots/584860042507/Approov-6e6cc7c6-7d53-418b-bf90-daf3f573db0a?versionId=1hBPS2APuHisTEu7jZj6utCyZFboUlaP&X-Amz-Security-Token=AgoJb3JpZ2luX2VjEH0aCXVzLWVhc3QtMSJGMEQCIFzPFX0PQOwdDNj1w%2BJIW7WXQh9rgk7E3qCBRQ9BaLBxAiA9iP0%2Bpl8WtFIUk%2FycrSFFPIADWciDVOt32MRKX5YUAyrjAwiG%2F%2F%2F%2F%2F%2F%2F%2F%2F%2F%2F%2F8BEAAaDDc0OTY3ODkwMjgzOSIMhexCK9EuoHHDPjtYKrcD8af1D1OTCICRa0s%2F8LbLKbGvO%2FuScVLlFtnpc76RHbdf4%2F4PGOGzASjCk2pXFXIcodBZfiAvD6pj6lWxkzB2cdYMx8uxc8x3P4X2%2BUu7%2FPBy8pRs%2BK2ZzFcYGmRlz8sriq%2BLTvepXEAi9n00ys69w%2FkyJRG5yhiqxisnQXnK1qM78s%2F2LMWhourE2y8hr9T0SFd6jt0g7GUL1O7aL%2BaDkxztBP4QAhWeQCzdJurPYSi2m%2FtutnDePC0hlz5vFcCch%2BDydEiv7My925J3UNJU54PmlQNxKkpLZsiklGUvTsSayEs2ejEw0ZVMdG8uiL6vPFd7ym3kbCFD5oKhiRHHW1z5DytvdiS0%2BQRLem6Mn%2BvMuG%2BqlKXxvhIwtgNwGETIqYoMc4TV9fvUDmy0SdD%2FDpt8xrXCGW9GsaR5TLK7CI4W8xBvqK9jFSyl3z7AUYrvFeZPv1GLx%2BGHo2o4dLBheU0bC6YQb8S%2FC6eETXRn4%2FB3tiHZBRRF5%2BF33P8KlNk0gEwMr3lhtEwN0OcbAGnb3CQEurDXAD9Atpo2D0EXJs4F3wW9svlzhp8pJxuGoQ0VdUmfnc1dVTDRi47nBTq1AT4s9wkOcUIKDjx%2B1FztYUJtC4fEoknP3nVDK4VfA2dIaDLOB6nCS8WCw8lluFeaECapK%2FcvIK5kfkMGStcXkhzGv0OBSnrcs97t1lZQYfbb4BnMAZxAfp%2BoPnUHbIHqU9nlFmlSj3jepVFym%2BtWJI8xEOtzNEs0T6EhdSpNw4XkM5X42%2FFN8mXKOONRS8RmcuO1%2FxHaLHFTN9eU1aqLTZ7YoDFFWO2N6w%2FWZ9d8%2Be8z1ASqIMQ%3D&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Date=20190521T061642Z&X-Amz-SignedHeaders=host&X-Amz-Expires=600&X-Amz-Credential=ASIA25DCYHY365WV25BP%2F20190521%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Signature=2abe4aa65b932fef7e2ae5c14c21b79a8a8c7984ebcbbf874fee0432a8100c9e"
  },
  "Configuration": {
    "TracingConfig": {
      "Mode": "PassThrough"
    },
    "Version": "$LATEST",
    "CodeSha256": "qB7Af0mH6kEmZw2dKLduP2HftUtarXnJho4vYFzzAxQ=",
    "FunctionName": "ApproovV2",
    "VpcConfig": {
      "SubnetIds": [],
      "VpcId": "",
      "SecurityGroupIds": []
    },
    "MemorySize": 128,
    "RevisionId": "8ded2bf5-787f-4079-8354-87e2bf39821e",
    "CodeSize": 75143,
    "FunctionArn": "arn:aws:lambda:us-east-1:584860042507:function:Approov",
    "Handler": "lambda_function.lambda_handler",
    "Role": "arn:aws:iam::584860042507:role/service-role/Approov-role-73dcj0fw",
    "Timeout": 3,
    "LastModified": "2019-04-22T15:09:03.172+0000",
    "Runtime": "python2.7",
```

```
        "Description": "Approov"
    },
    "Tags": {
        "Environment": "Approov"
    }
}
lambda@host:~$
```

The zip will be downloaded unpack the zip.

```
lambda@host:~/tmp$ mv ApproovV2-6e6cc7c6-7d53-418b-bf90-daf3f573db0a  Approov.zip
lambda@host:~/tmp$ ls
Approov.zip
lambda@host:~/tmp$ unzip Approov.zip
Archive:  Approov.zip
   creating: ipaddress-1.0.16.dist-info/
  inflating: ipaddress-1.0.16.dist-info/INSTALLER
  inflating: ipaddress-1.0.16.dist-info/DESCRIPTION.rst
  inflating: ipaddress-1.0.16.dist-info/METADATA
  inflating: ipaddress-1.0.16.dist-info/WHEEL
  inflating: ipaddress-1.0.16.dist-info/RECORD
  inflating: ipaddress-1.0.16.dist-info/top_level.txt
  inflating: ipaddress-1.0.16.dist-info/pydist.json
  inflating: ipaddress.py
  inflating: ipaddress.pyc
   creating: jwt/
  inflating: jwt/algorithms.pyc
  inflating: jwt/__main__.py
  inflating: jwt/api_jws.pyc
  inflating: jwt/__init__.pyc
  inflating: jwt/api_jwt.pyc
  inflating: jwt/__init__.py
  inflating: jwt/api_jws.py
  inflating: jwt/compat.py
  inflating: jwt/__main__.pyc
  inflating: jwt/exceptions.py
   creating: jwt/contrib/
  inflating: jwt/contrib/__init__.pyc
```

```
 extracting: jwt/contrib/__init__.py
   creating: jwt/contrib/algorithms/
  inflating: jwt/contrib/algorithms/py_ecdsa.py
  inflating: jwt/contrib/algorithms/__init__.pyc
 extracting: jwt/contrib/algorithms/__init__.py
  inflating: jwt/contrib/algorithms/pycrypto.py
  inflating: jwt/contrib/algorithms/pycrypto.pyc
  inflating: jwt/contrib/algorithms/py_ecdsa.pyc
  inflating: jwt/algorithms.py
  inflating: jwt/api_jwt.py
  inflating: jwt/exceptions.pyc
  inflating: jwt/compat.pyc
  inflating: jwt/utils.pyc
  inflating: jwt/utils.py
  inflating: lambda_function.py
   creating: PyJWT-1.4.2.dist-info/
  inflating: PyJWT-1.4.2.dist-info/INSTALLER
  inflating: PyJWT-1.4.2.dist-info/metadata.json
  inflating: PyJWT-1.4.2.dist-info/DESCRIPTION.rst
  inflating: PyJWT-1.4.2.dist-info/entry_points.txt
  inflating: PyJWT-1.4.2.dist-info/METADATA
  inflating: PyJWT-1.4.2.dist-info/WHEEL
  inflating: PyJWT-1.4.2.dist-info/RECORD
  inflating: PyJWT-1.4.2.dist-info/top_level.txt
lambda@host:~/tmp$
lambda@host:~/tmp$ vi lambda_function.py
```

Modify the lambda_function.py file with the new secret.

Save the file, Archive the existing function, by copying it out of the current directory, then repackage the Function, call the zip file Approov1.zip

```
lambda@host:~/tmp$ zip -r ../Approov1.zip .
  adding: ipaddress-1.0.16.dist-info/ (stored 0%)
  adding: ipaddress-1.0.16.dist-info/WHEEL (stored 0%)
  adding: ipaddress-1.0.16.dist-info/METADATA (deflated 55%)
  adding: ipaddress-1.0.16.dist-info/top_level.txt (stored 0%)
```

```
adding: ipaddress-1.0.16.dist-info/INSTALLER (stored 0%)
adding: ipaddress-1.0.16.dist-info/RECORD (deflated 41%)
adding: ipaddress-1.0.16.dist-info/pydist.json (deflated 50%)
adding: ipaddress-1.0.16.dist-info/DESCRIPTION.rst (deflated 2%)
adding: lambda_function_production.py (deflated 70%)
adding: jwt/ (stored 0%)
adding: jwt/contrib/ (stored 0%)
adding: jwt/contrib/__init__.pyc (deflated 22%)
adding: jwt/contrib/__init__.py (stored 0%)
adding: jwt/contrib/algorithms/ (stored 0%)
adding: jwt/contrib/algorithms/py_ecdsa.py (deflated 60%)
adding: jwt/contrib/algorithms/__init__.pyc (deflated 21%)
adding: jwt/contrib/algorithms/__init__.py (stored 0%)
adding: jwt/contrib/algorithms/py_ecdsa.pyc (deflated 53%)
adding: jwt/contrib/algorithms/pycrypto.py (deflated 58%)
adding: jwt/contrib/algorithms/pycrypto.pyc (deflated 55%)
adding: jwt/__init__.pyc (deflated 40%)
adding: jwt/__init__.py (deflated 45%)
adding: jwt/utils.py (deflated 61%)
adding: jwt/api_jwt.pyc (deflated 58%)
adding: jwt/__main__.py (deflated 65%)
adding: jwt/api_jws.pyc (deflated 57%)
adding: jwt/utils.pyc (deflated 55%)
adding: jwt/algorithms.pyc (deflated 66%)
adding: jwt/compat.py (deflated 54%)
adding: jwt/exceptions.py (deflated 69%)
adding: jwt/api_jwt.py (deflated 73%)
adding: jwt/api_jws.py (deflated 73%)
adding: jwt/compat.pyc (deflated 44%)
adding: jwt/exceptions.pyc (deflated 74%)
adding: jwt/__main__.pyc (deflated 45%)
adding: jwt/algorithms.py (deflated 79%)
adding: PyJWT-1.4.2.dist-info/ (stored 0%)
adding: PyJWT-1.4.2.dist-info/WHEEL (deflated 14%)
adding: PyJWT-1.4.2.dist-info/METADATA (deflated 59%)
adding: PyJWT-1.4.2.dist-info/entry_points.txt (deflated 5%)
adding: PyJWT-1.4.2.dist-info/top_level.txt (stored 0%)
adding: PyJWT-1.4.2.dist-info/INSTALLER (stored 0%)
adding: PyJWT-1.4.2.dist-info/RECORD (deflated 48%)
adding: PyJWT-1.4.2.dist-info/metadata.json (deflated 55%)
```

```
  adding: PyJWT-1.4.2.dist-info/DESCRIPTION.rst (deflated 54%)
  adding: lambda_function_logging.py (deflated 70%)
  adding: ipaddress.pyc (deflated 73%)
  adding: lambda_function.py (deflated 70%)
  adding: ipaddress.py (deflated 81%)
lambda@host:~/tmp$
```

Now upload the binary file (zip) as a code replacement for the Approov function, this version will now become the $LATEST version.

```
lambda@host:~/tmp$ aws lambda update-function-code --function-name Approov1 --zip-file
fileb://Approov1.zip
{
    "FunctionName": "Approov1",
    "LastModified": "2019-05-21T06:42:59.599+0000",
    "RevisionId": "8de347aa-83f6-42be-8062-73bfe2d4c6d5",
    "MemorySize": 128,
    "Version": "$LATEST",
    "Role": "arn:aws:iam::584860042507:role/service-role/Approov1-role-73dcj0fw",
    "Timeout": 3,
    "Runtime": "python2.7",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "frX5TUq9bS8jJwub215wNC/rUYcbgOv5Y9DOfi8p49k=",
    "Description": "Approov",
    "VpcConfig": {
        "SubnetIds": [],
        "VpcId": "",
        "SecurityGroupIds": []
    },
    "CodeSize": 39089,
    "FunctionArn": "arn:aws:lambda:us-east-1:584860042507:function:Approov1",
    "Handler": "lambda_function.lambda_handler"
}
lambda@host:~/tmp$
```

NOTE: *These are just a small subset of commands used to configure, modify and manage lambda function via the AWS CLI.*

If you are using an environment variable for your secret, instead of embedding the secret inside the function, then management is even simpler, all that needs to be completed is for you to manage the lambda function's actual configuration.

For Example, modifying the SECRET where the secret is stored as an environment variable:

Where we want to modify the existing SECRET to "kasjgadsasdjhgjhgadsf"

```
lambda@host:~$ aws lambda update-function-configuration --function-name Approov1 --environment
Variables={"SECRET"="kasjgadsasdjhgjhgadsf"}
{
    "Runtime": "python2.7",
    "VpcConfig": {
        "SecurityGroupIds": [],
        "VpcId": "",
        "SubnetIds": []
    },
    "RevisionId": "481fb097-87c6-44bc-98c0-549e4b0d5dba",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "frX5TUq9bS8jJwub215wNC/rUYcbgOv5Y9DOfi8p49k=",
    "FunctionName": "Approov1",
    "Timeout": 3,
    "Handler": "lambda_function.lambda_handler",
    "MemorySize": 128,
    "Description": "Approov",
    "Environment": {
        "Variables": {
            "SECRET": "kasjgadsasdjhgjhgadsf"
        }
    },
    "CodeSize": 39089,
    "FunctionArn": "arn:aws:lambda:us-east-1:584860042507:function:Approov1",
    "Version": "$LATEST",
    "LastModified": "2019-05-21T09:18:54.811+0000",
    "Role": "arn:aws:iam::584860042507:role/service-role/Approov-role-73dcj0fw"
}
lambda@host:~$
```

The functions secret is now permanently changed.

CLOUDFORMATION

CloudFormation is an AWS Service for building your infrastructure via configuration data, this is known as Infrastructure As Code (IaC).

Other alternative build systems for our serverless solution are Terraform, and SAM (Serverless Application Model) .

Some continuous deployment systems such as Puppet, Chef and Ansible are also now capable of building infrastructure.

You can manage, provision, update, modify and version all of your AWS resources through CloudFormation.

The below is an (subset) example of a CloudFormation stack for building and deploying a proxy API gateway service with the Approov authoriser, this example only shows a subset of the resources and dependencies required to build a best-practices AWS environment.

Parameters, these are variable settings for settings that should not be hardcoded and are referenced more than once throughout the template.

We only have two:

apiGatewayStageName set to ”Approov”, and lambdaFunctionName set to “Approov”

The CloudFormation code for this is as follows:

```
Parameters:
  apiGatewayStageName:
    Type: "String"
    AllowedPattern: "^[a-z0-9]+$"
    Default: "Approov"

  lambdaFunctionName:
    Type: "String"
    AllowedPattern: "^[a-zA-Z0-9]+[a-zA-Z0-9-]+[a-zA-Z0-9]+$"
    Default: "Approov"
```

Resources section sets up and configures individual parts of the solution built by CloudFormation, it will also use the variable parameters (if any) in the Parameter section.

These resources can reference each other, some can only be built if other parts already exist or have been prebuilt.

CloudFormation is also sophisticated enough to build services required that are actually required (dependencies) by other services.

An example of API Gateway with regards to Approov, is the following, which sets out the API stage, authoriser and lambda function used….

```
Resources:
  apiGateway:
    Type: "AWS::ApiGateway::RestApi"
    Properties:
      Name: "Proxy"
      Description: "Full Proxy API"
apiGatewayRootMethod:
    Type: "AWS::ApiGateway::Method"
    Properties:
      AuthorizationType: "CUSTOM"
      authorizerId: "1h6spz"
      HttpMethod: "ANY"
      Integration:
        IntegrationHttpMethod: "ANY"
        Type: "AWS_PROXY"
        Uri: !Sub
          - "arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-
31/functions/${lambdaArn}/invocations"
          - lambdaArn: !GetAtt "lambdaFunction.Arn"
      ResourceId: !GetAtt "apiGateway.RootResourceId"
      RestApiId: !Ref "apiGateway"

  apiGatewayDeployment:
    Type: "AWS::ApiGateway::Deployment"
    DependsOn:
      - "apiGatewayRootMethod"
    Properties:
      RestApiId: !Ref "apiGateway"
      StageName: !Ref "apiGatewayStageName"

  lambdaFunction:
    Type: "AWS::Lambda::Function"
    Properties:
      Code: {
          "S3Bucket": "approovproxyfunctions",
          "S3Key": "                    "
      },
      Description: "Approov JWT Validator"
      FunctionName: !Ref "lambdaFunctionName"
```

```
Handler: "index.handler"
MemorySize: 128
Role: !GetAtt "lambdaIAMRole.Arn"
Runtime: "python2.7"
Timeout: 10
```

Once all relevant resources have been successfully inserted, upload the file either via console or the AWS CLI into the CloudFormation service and create the CloudFormation stack.

DATA LOGGING

CLOUDWATCH

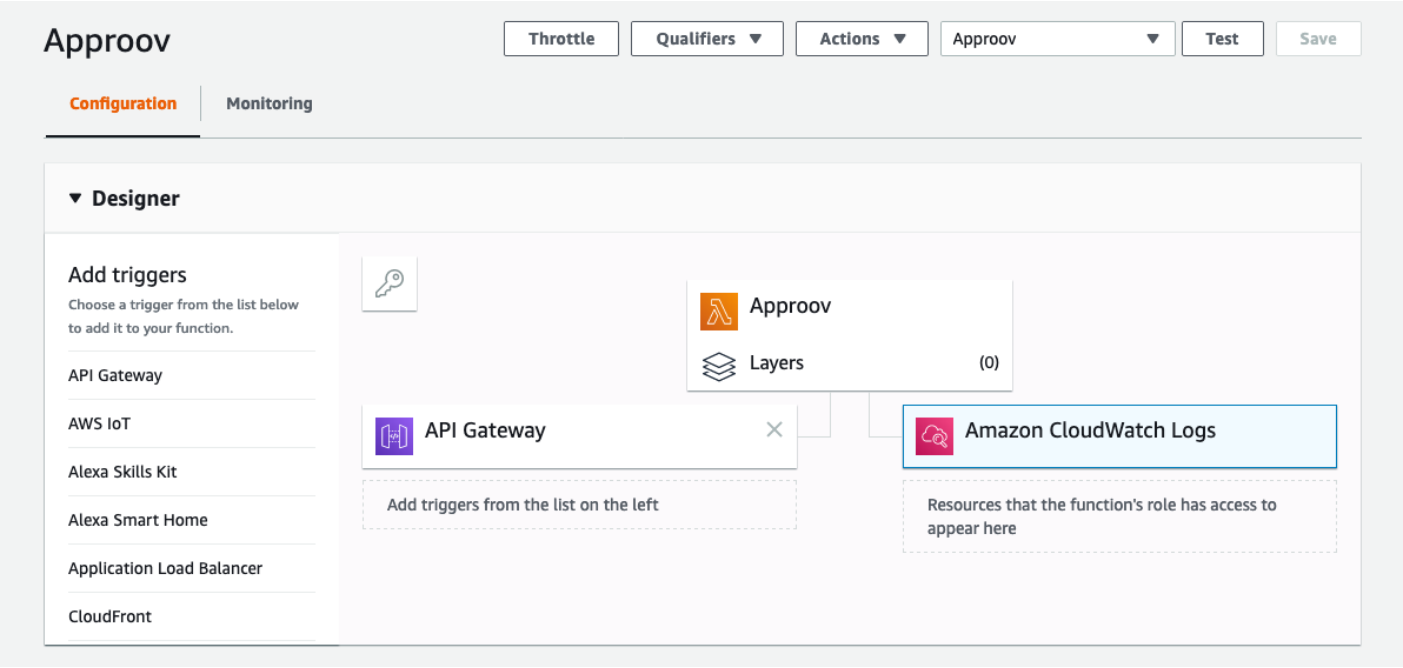
CloudWatch logging is enabled automatically for your lambda function executions.

The data is stored for 2 weeks and (without customisations) comes at no charge.

Our Lambda Approov function, on every execution automatically sends execution data to CloudWatch.

When we first created the Approov function, CloudWatch for our function was automatically enabled.

For example, within the Approov Function Configuration window (Lambda->Functions->Approov).



Log entries from individual lambda executions are then sent to streams within your LogGroup.

For example:

Go to CloudWatch->Logs

Approov's log entries are stored inside /aws/apigateway/Approov, Click on the highlighted name.

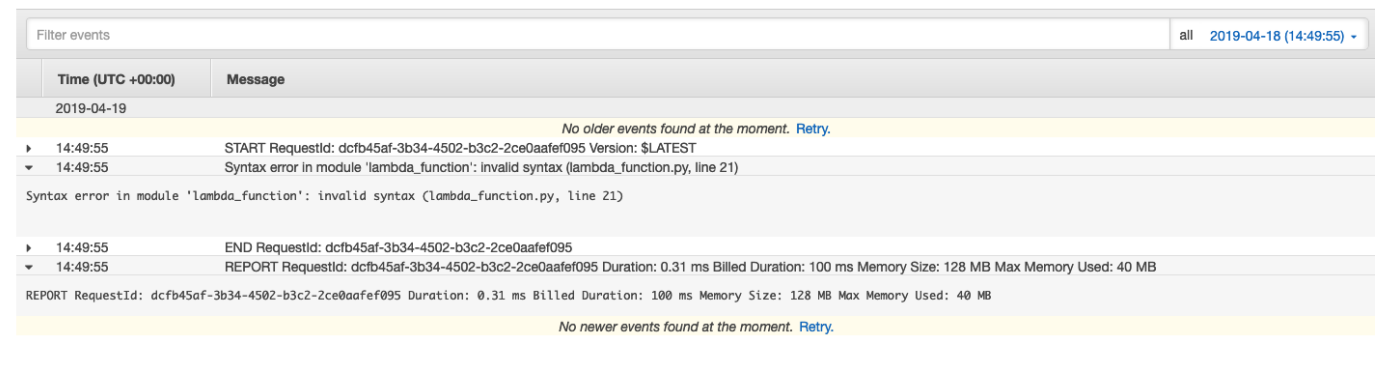
Most people will expect to see one log with all the transactions inside it.

Within my LogGroup i have 301 "Log streams".

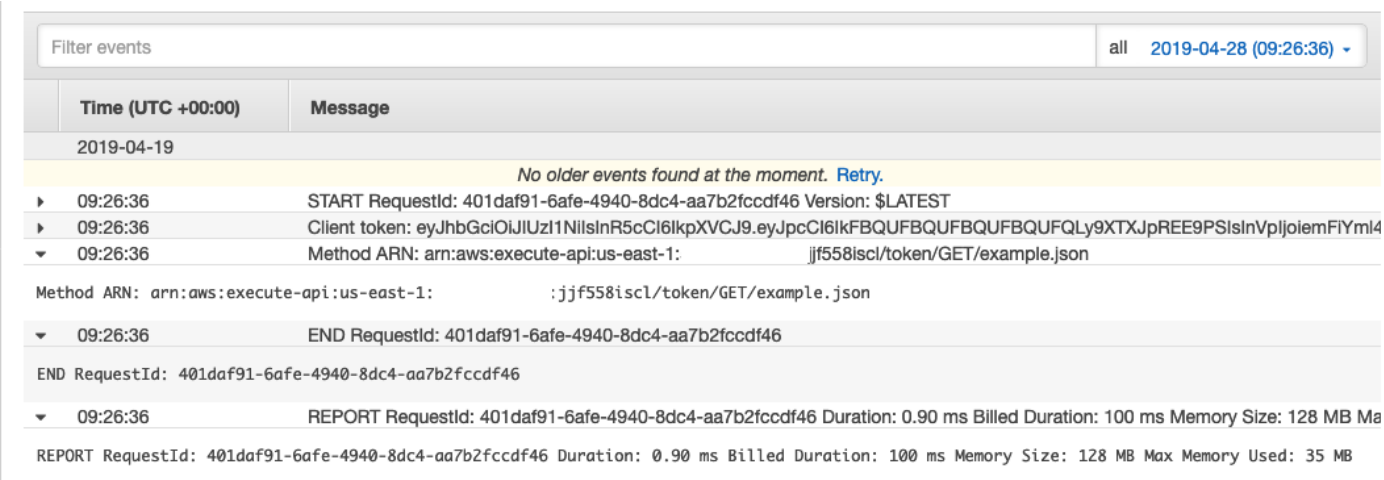
This is where it gets interesting. These streams are basically individual logs for ALL the AWS microcontainers that are available to run your function. Under a heavy workload, there will be many more.

Each execution will log different information depending upon the execution success/failure, validation, data returned, or access denied, etc.

An example of a coding error within the function:



Example of a successful execution and data was returned:



Notice the 90ms, this should be around 20-30ms, this is because the function has been "Cold started" this is where the first execution takes longer just due to the resources needed to be created to run the function.

Any other execution after this will return to about 20-30ms. If there are no executions for approximately 7 minutes in a low traffic environment the function will require cold-starting again, in higher traffic environments (consecutive executions) this “resource availability” number has been seen to span out to almost 40 minutes.

This is important as billing for lambda executions is in 100ms periods, if lambda consistently takes 105ms to run it will cost twice as much as a 95ms function.

GENERATING ACCESS LOGS

If your API data cache is enabled, and you want or need to store and retrieve the logging data in a similar structure to what your origin may produce then you will need to configure access logging. This is due to the issue where the origin will only receive a small number of the overall requests that are serviced by API gateway and its inbuilt cache.

If you do not cache data requests, and all incoming validated requests are forwarded to the origin, then there is no need to log access requests if we disable the data cache inside API Gateway stage.

NOTE: *401 and 403 entries will not be forwarded to the origin, they will however appear in any access logs and standard cloudwatch logs within AWS*.

With “Use HTTP Proxy integration” automatically enabled, inside the API Integration request section, all data from the validated request, including headers are proxied to the origin.

Customising the log format on the origin will enable the ip of the client and the user-agent to be collected in each log entry. We can make use of the following headers are proxied to the origin

- x-forwarded-for, this is our client ip address
- user-agent, this is the browser type.
-

An example of a log entry an nginx origin, with the following custom format:

```
$http_x_forwarded_for $remote_addr [$time_local] "$request" $status $body_bytes_sent "$http_referer" "$http_user_agent";
```

will result in the following entry:

```
88.202.226.12 3.216.143.35 [06/Jul/2019:18:24:12 +0000] "GET /example.json HTTP/1.1" 200 873 "curl/7.35.0"
```

However, as previously stated, only successfully validated requests will be returned to the origin, denials (403 and 401 returned requests) are only logged into CloudWatch standard logs and, and if configured, CloudWatch access logs.

If you want to create or format log entries for logging information similar to apache style logging, there is a feature within the stage configuration to create these log entries

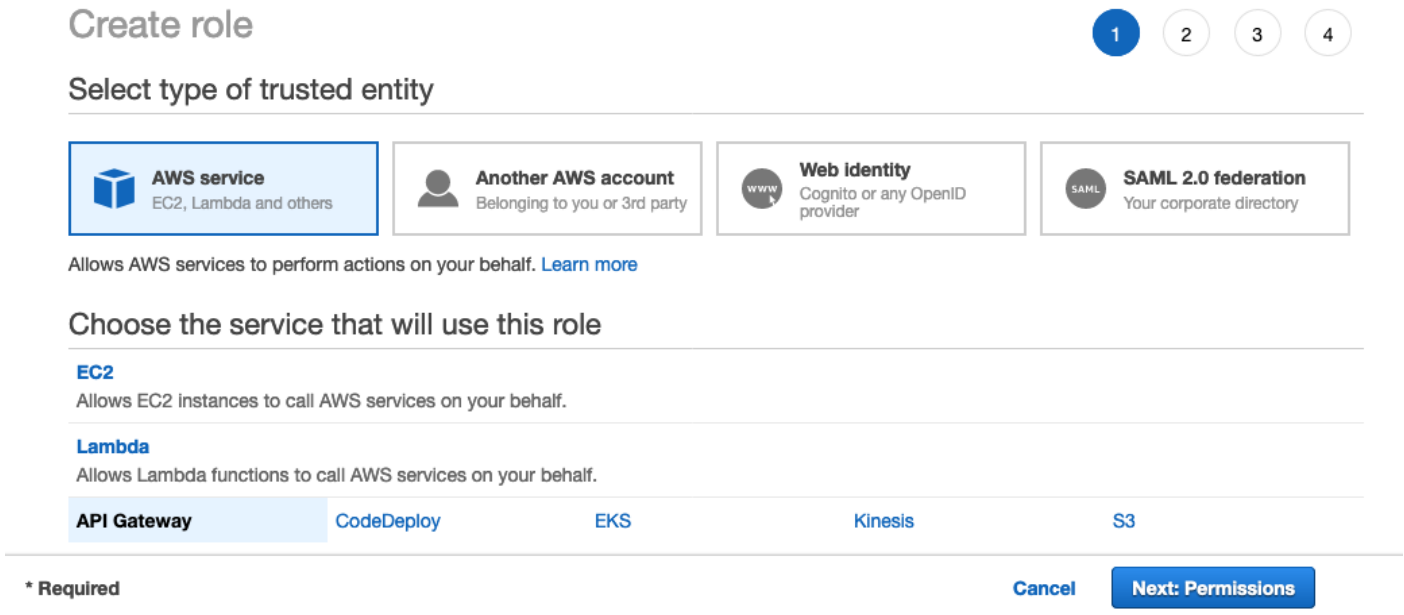
We will create a custom logGroup to store these log entries.

Within the deployed stage, access CloudWatch logs and/or Custom Access logs can be enabled.

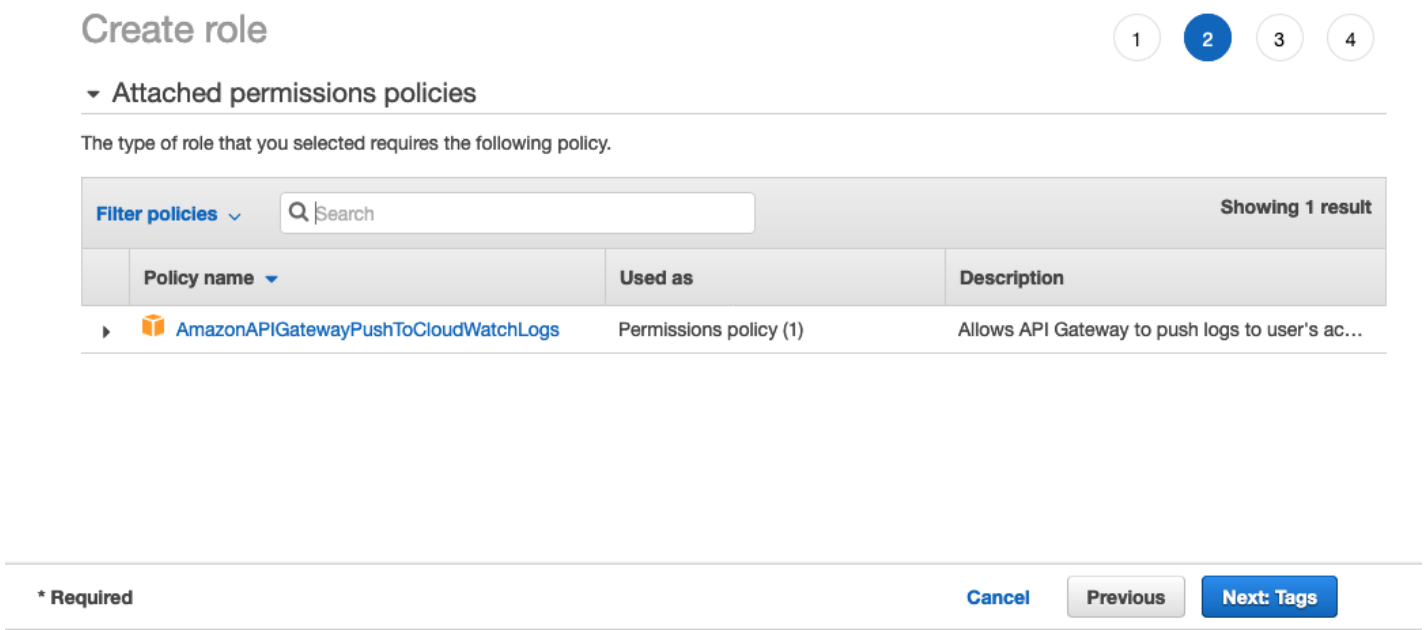
These log entries are configured within the Log format, you will need a Cloudwatch Log Group to send these log entries to, and we also need an IAM role to allow API Gateway to access these logs.

Go to IAM->Roles, Create Role.

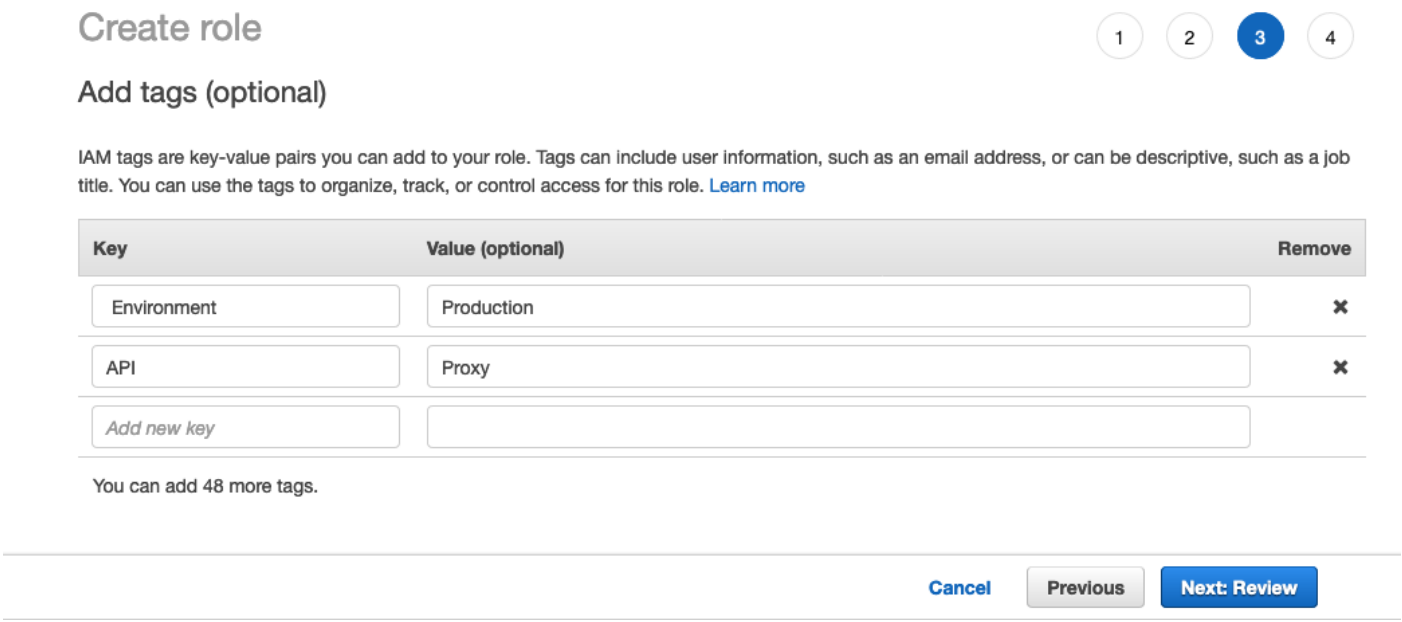
Under “AWS Service”, select “API Gateway”, select “Next Permissions”.



The permission should be autofilled in the window.

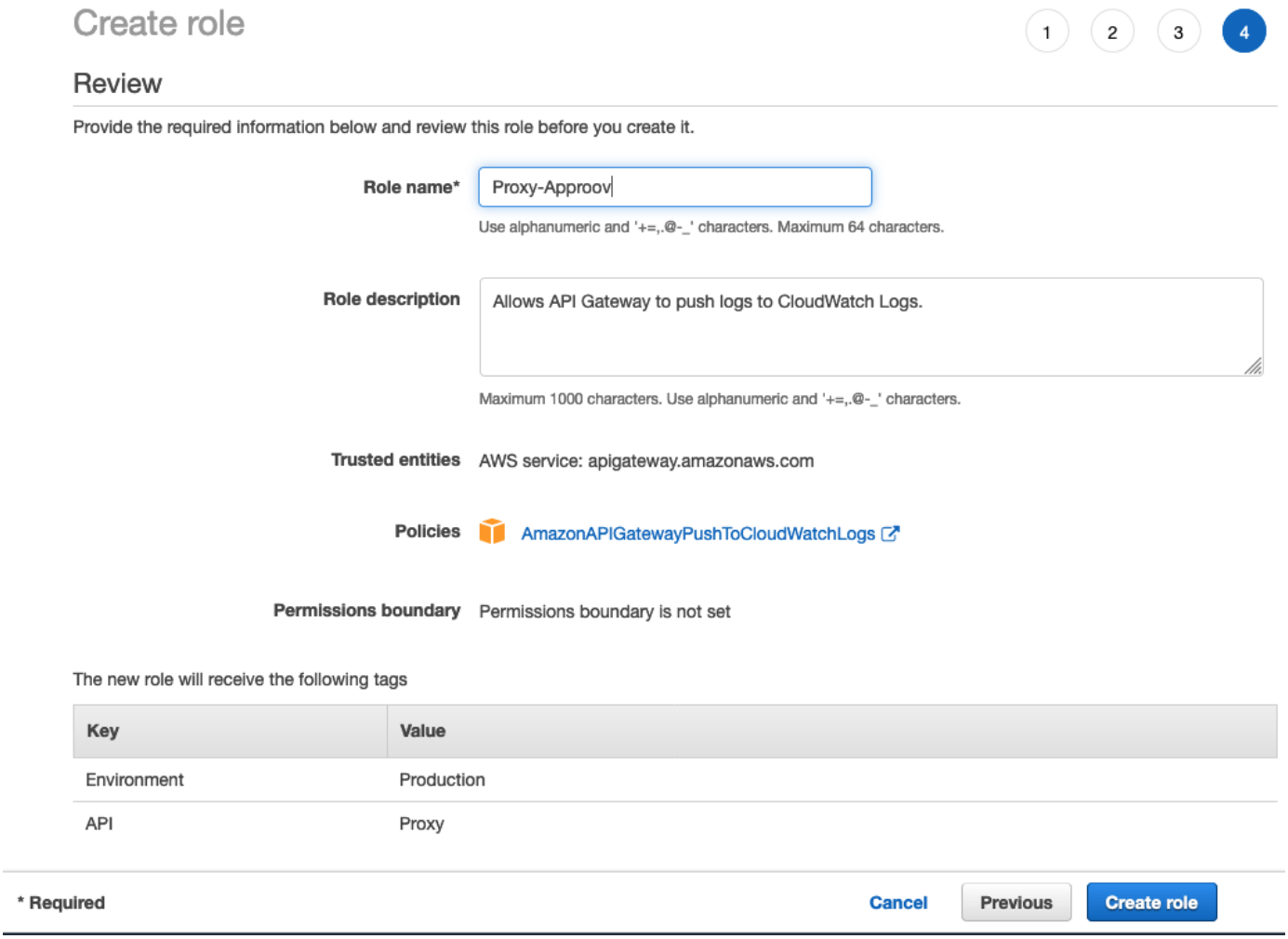


Click “Next:Tags”



I’ve filled in two [optional] tags, as examples, click “Next: Review”.

Give your role a name, this will only be used for API Proxy, Approov stage, this role can have other permissions added to it later, so that API Proxy,Stage:Approov, can have access to other specific AWS services that this particular API stage requires, We should always keep to “Least Privilege” when concerning access and permissioning.



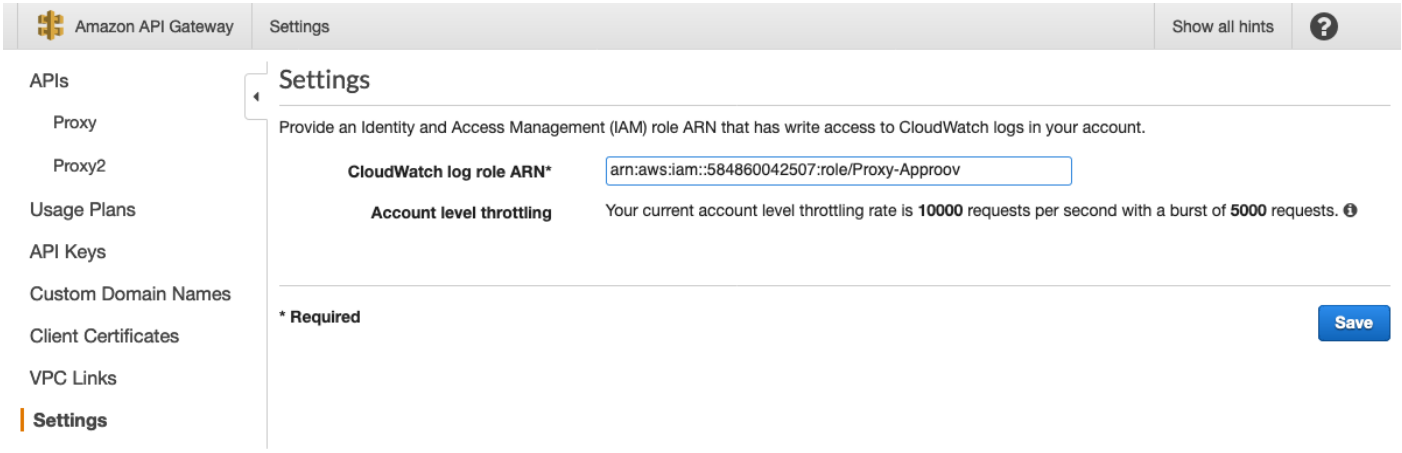
Click “Create Role”

Now Click on the Approov-Proxy role.

And copy the Role ARN, as below.

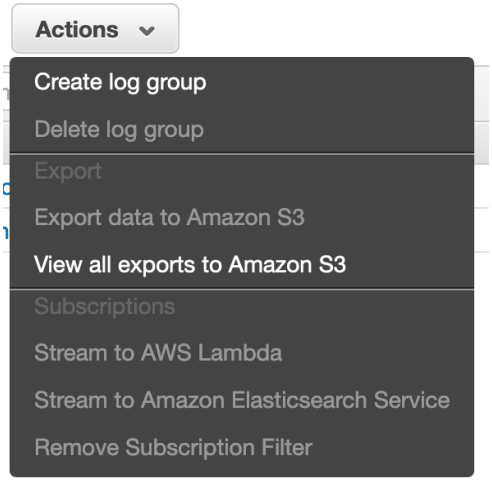
Now go to API Gateway->Settings.

Paste you Role ARN into the field.



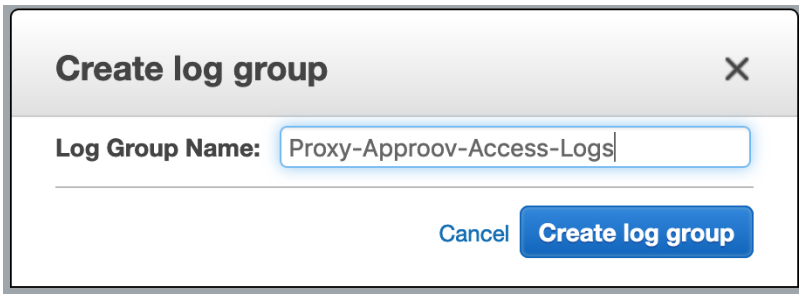
Click Save.

Then go to CloudWatch->Logs->Actions->create log group.



We'll call our CloudWatchh Log Group Proxy-Approov-Access-Logs

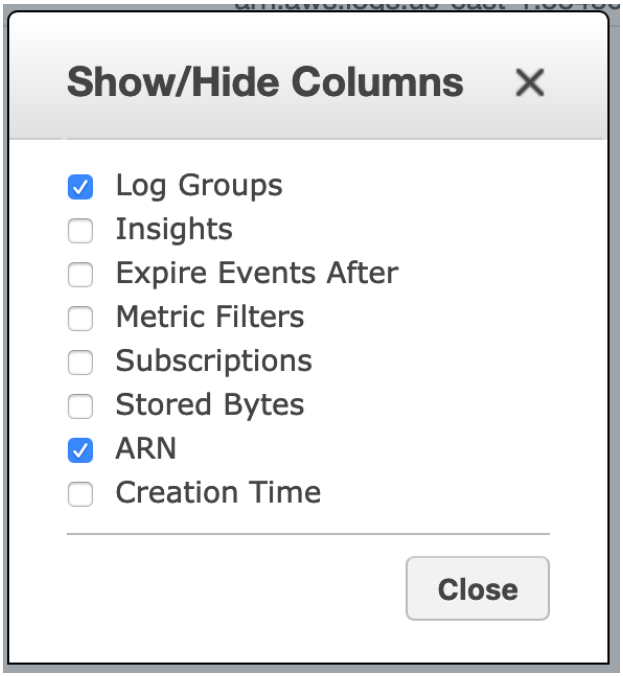
<API>-<STAGE>-<LOGTYPE>-Logs



We will not yet create any log metrics

Now at the right handside of the window you will see the setup-wheel, , click on this to reveal the Column choices, we need to know the LogGroup ARN for the Access log configuration.

Check the Log Group, and ARN, uncheck the rest.



Click close.

Now, Copy the ARN value.

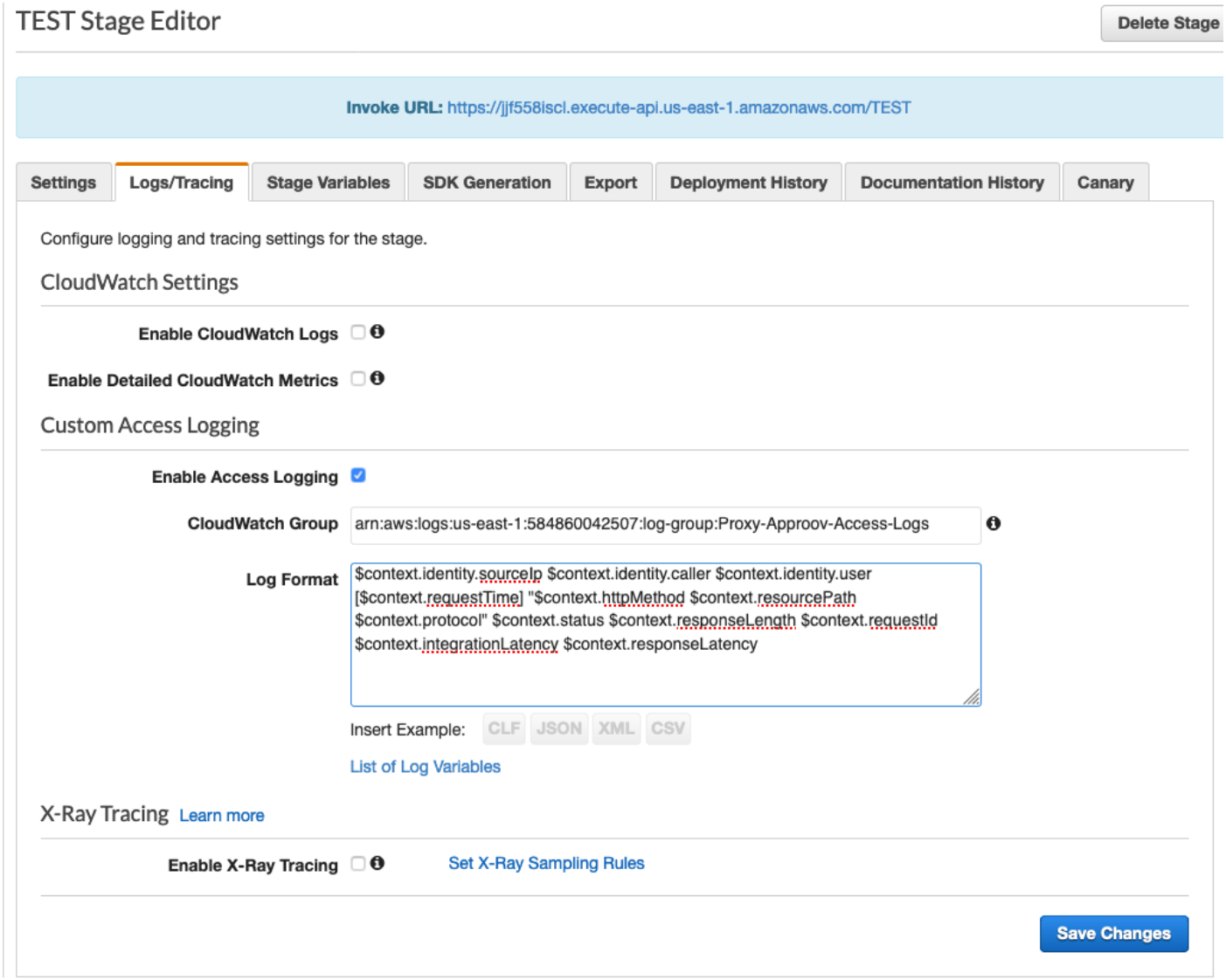
We need to create an IAM role to give API gateway access to our CloudWatch Logs

So go to API Gateway -> Proxy -> Stage -> TEST> Logs/Tracing

We will not enable full CloudWatch logging, we will only use access logging.

Check “Enable Access Logging”, paste in your new Access LogGroup CloudWatch ARN you copied previously when you created it, within the CloudWatch Group field BUT without the “:*” on the end of the ARN.

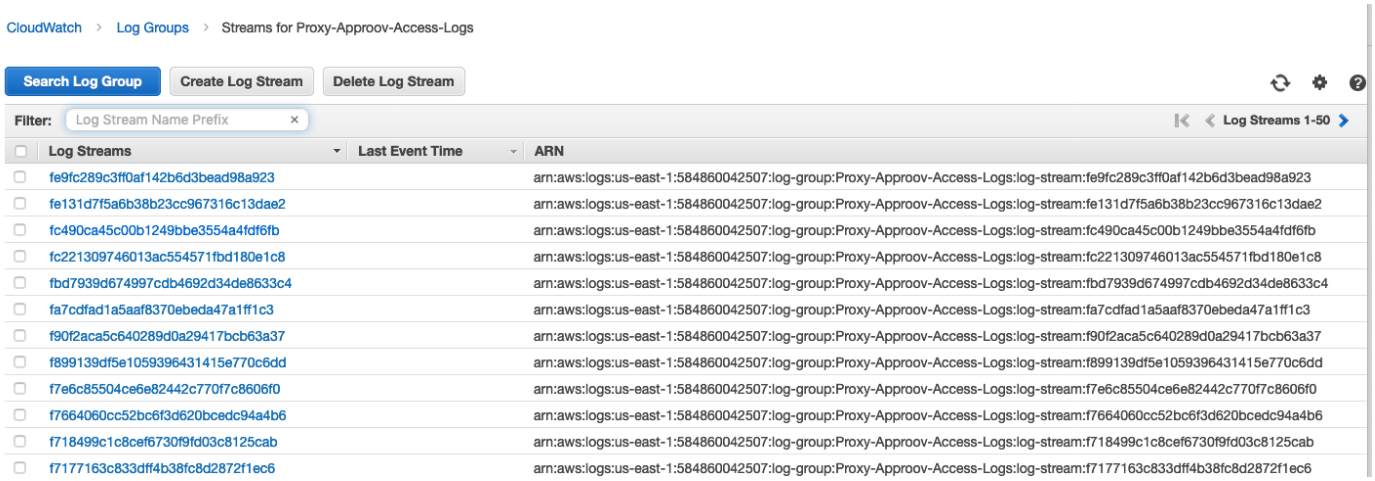
I will just use the example CLF, but with $context.integrationLatency $context.responseLatency added.



Click “Save Changes”.

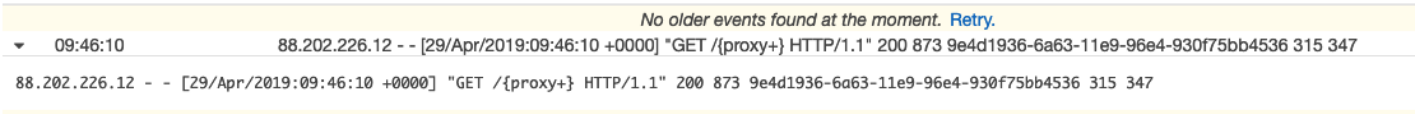
Lambda is described as a “Serverless” function, but as with everything in the cloud, it has to actually run on hardware/software resources somewhere, in CloudWatch you will see logs from these systems.

Now go into CloudWatch and into the logGroup that has just been created.



NOTE: *Generally only a small number of these will be used, depending on how many consecutive functions are being executed at the same time, for a low volume API it will be less than half a dozen.*

Our test transaction has been successfully recorded:

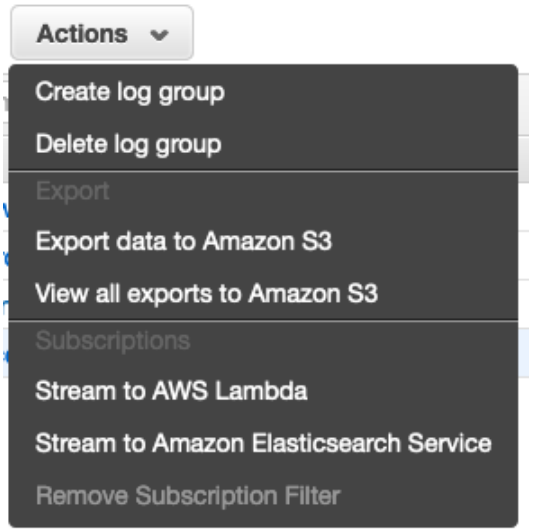


There are options to stream the data to AWS ElasticSearch Service or stream the data to a lambda function (similar to a trigger), where the data can be proxied to another AWS service or even an external system (but look out for data transfer costs).

You can create these at LogGroup level .

Click on Proxy-Approov-Access-Logs, within CloudWatch->logs

Click on the Actions button to see the options associated with this logGroup.



There are three options that you can use to “funnel” the streams data to a single endpoint for processing.

EXPORTING LOG DATA

Export data to S3

Why? So you can either download it for external processing or, use other AWS services like AWS Athena to process the structured data stored on S3 for a report, or better, get Athena to export all the data into a single unified timestamped streamed file (treating the data from the streams like a data lake).

Stream to AWS Lambda

Lambda can take the transaction (request entry) and process it, send it into AWS Kinesis, Or even a rsyslog server. The Lambda function will have to be small and tight to maximise performance as every entry will execute a function. Understanding your data estimates is important as data transfer costs could be, in some circumstances, considerable.

Stream to Elasticsearch Service (ES)

Probably the most used service I've come across, out of all 3 options.

ES Allows real time data ingression (via logstash), realtime processing(ElasticSearch), and real time reports (via Kibana) on requests and token validations

Re-formatted data can be exported from ES into S3.

LOGGING WITHIN LAMBDA

Building logging within the lambda function itself is another way of inserting custom formatted error or information into CloudWatch logs.

These entries will be written to the CloudWatch streams associated with the LogGroup setup within CloudWatch logs.

In Python you only need to import the standard logging library.

This allows customisation of what information goes into the log entries.

The simplest way of creating an entry is the following:

```
import logging
logger = logging.getLogger()
logger.setLevel(logging.INFO)
logger.error('something went wrong')
```

Using the logging library, lambda can (and does) create logger handlers for use in taking the log transaction and formatting it for CloudWatch logs, this can and does produce a lot of unwanted data.

You can override this “functionality” by deleting the default format handlers and resetting a single one to your own, via “logging.basicConfig”.

For example, By setting the the log format as follows:

```
FORMAT= '%(asctime)s %(clientip)s %(message)s'
problem = {'clientip': '123.45.67.89', 'message': http.status }

logger = logging.getLogger()
if logger.handlers:
    for handler in logger.handlers:
        logger.removeHandler(handler)
logging.basicConfig(format=FORMAT,level=logging.DEBUG)
logger.warning(Token problem: %s', 'Invalid signature", extra=problem)
```

Using Logger inside your function will send the entry to the CloudWatch log group named after the Lambda function being executed, in this case the log group will be: /aws/lambda/approov

This creates an entry similar to:

```
2019-04-08 20:12:09, 123.45.67.89 403 Token problem: Invalid signature
```

GENERATING LONG DATED TOKENS

Long dated tokens, whose expiry can last years, are beneficial for environments where attestation is not possible, these are mainly used for Development and static systems that need constant access to the Approov protected API. They are useful in the following situations:

- Monitoring services
    - where you want a monitoring system like nagios, zabbix or Prometheus, to scrape the same data as available to your genuine mobile apps, 24x7, for example to check data returns for incorrect data structures or possible issues with structured requests.
- Apps executed in a privileged environment.
    - I have generated these for developers to use within their apps when the app is either running inside a virtualized environment or just running within debug or privileged mode.
- B2B services using the same API
    - I've also encountered environments where server to server integration was required.

This is where client data is commercially "wholesaled" to a business partner, and the data source is through the same API, or where the B2B environment requires more finer security than an API key.

Approov V2 has the ability to generate long dated tokens that can have a time expiry measured in years.

The method to manage and generate long dated tokens is through the Approov CLI Tool.

Approov Tool

The “Approov Tool” is a command line interface (CLI) that is used to manage your Approov tokens.

You will need an existing Approov account and an administration token to create and manage all tokens associated with your account.

If you do not currently have a Approov account, go here to sign up for a free monthly trial.

Once signed up, you will be sent a time limited link to down a zip file that contains:

- Administration Token
- Development Token.

You also need to download and install the latest version of the Approov Tool, here.

This downloaded zip file contains the CLI version for MacOS, Linux and Windows.

Please refer to Approov’s online Installation instructions for MacOS, Linux and MS Windows.

Once the Approov CLI has been successfully installed, you can create a long dated token:

To configure the Approov CLI (my example is Linux) to use the management token, load the administration token into an environment variable “APPROOV_MANAGEMENT_TOKEN”.

```
# export APPROOV_MANAGEMENT_TOKEN=`cat administration.tok`
#
# echo $APPROOV_MANAGEMENT_TOKEN
eyJhbGciOiJIUzI1NiIsInR5cCI6IkPXVCJ9.eyJleHU0OjI7MjgwND7ZODIsImlzcyI6ImFwcHovb3YiLCJlb
WwiOiJzdGVwaGVuQGF1dG9ub21hLnN5c3RlbXMiLCJ1c3IiOiJhdXRvbm9tYSIsImNzdCI6ImF1YG9u
b21hIiwicmxzI7oibWRhdCIsInVybCI6ImxpZmZleS1ldzEuYXBwcm9vdIOuaW8ifQ.iLmmHOFhih6nnyDj
N0Kw7FIAybrFvmd5YM1ytu4-E1s
#
```

NOTE: *If you have not previously configured this variable into your default shell profile, it would be advisable to do so.*

Now, configure your API, by adding the API gateway custom domain into Approov. I will add in api.dataproxy.io.

Since we are configuring Approov as the first API, we can just type YES.

```
# approov api -add api.dataproxy.io
WARNING: adding the API will have an immediate impact on your apps in production.
If you wish to continue then please enter YES and return: YES
added new API domain api.dataproxy.io with pin
N6Zn5Mpd3d6rLduZi205wjax03VL3oYQ+iTtVkTyOi4=
#
```

Now create a long dated token, with a unique name (issuer) and an expiry, the expiry can be years, but the token needs to have an expiry less than the admin token's expiry, this means the date needs to be less than 25 years.

I'll call mine "Monitor" and give it an expiry of 10 years.

```
# approov token -genLongLived Monitor,10y
token will expire at 2029-07-24 15:33:48
WARNING: Long lived token should never be integrated into public clients and their security must be carefully managed
eyJhbGciOiJIUzI1NiIsInY7cCI6IkpXVCJ9.eyJleHAiOjE4Nzk1OTgwMjgsImlzcyI6Ik1vbml0b3IifQ.rTNTZfkRTuF-kak3zi7eNKlC09qrTMBs3ywIupu-TPo
```

Check the token to make sure it's valid.

```
# export token=eyJhbGciOiJIUzI1NiIsInY7cCI6IkpXVCJ9.eyJleHAiOjE4Nzk1OTgwMjgsImlzcyI6Ik1vbml0b3IifQ.rTNTZfkRTuF-kak3zi7eNKlC09qrTMBs3ywIupu-TPo
#
# approov token -check $token
valid: JWS {"exp":1879598182,"iss":"Monitor"}
```

Now you can use this token for testing your API, in Postman, curl and inside the Console.

The API can also return the base64 encoded Secret.

You need the Approov Secret for the section on configuring the token authoriser.

To get the secret from Approov, simply run the following command:

```
# approov secret -get
JkQZGIwcw1m9DDfkyTKI5XCy20oTgGT6D
```

EXTERNAL TOOLSETS

The following toolsets can be used to check the request return, and any errors on your new API.

There are two main toolsets that I use, first is Postman a free GUI application (with premium paid advance features) this was a free chrome browser plugin, now it is a fully featured application available on windows, linux and macos.

The other is curl available for just about all Unix and Linux variants, 32 and 64 bit Windows (available as a MSI) and macos (via homebrew and others).

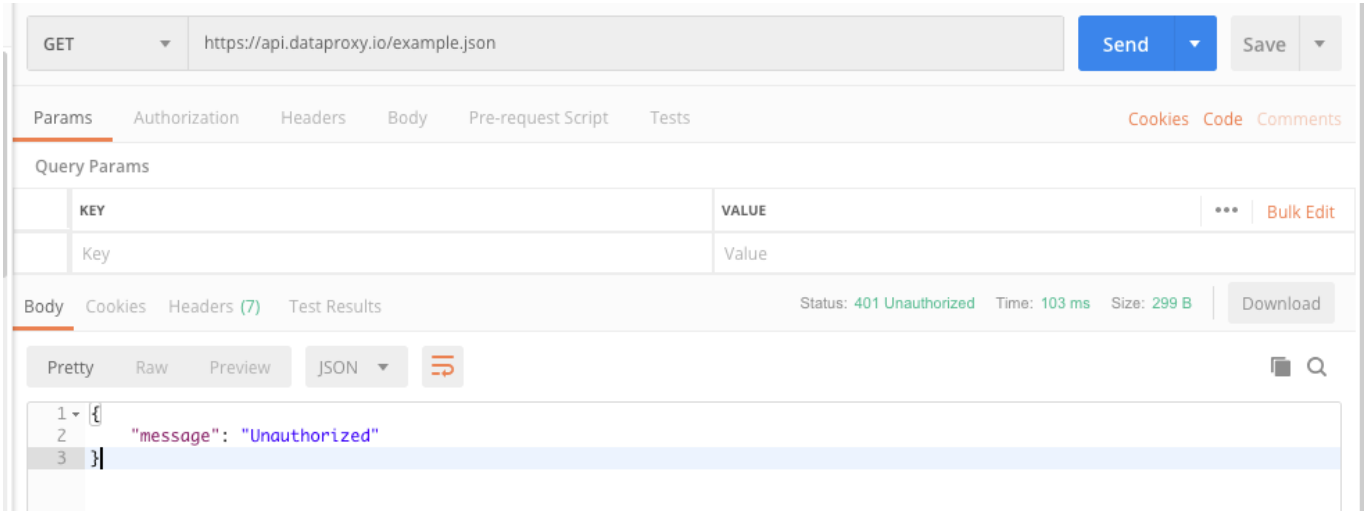
POSTMAN

Download Postman for your operating system from here

Install it and run it.

Place your test URL for your API gateway custom domain in the GET field box, and press send.

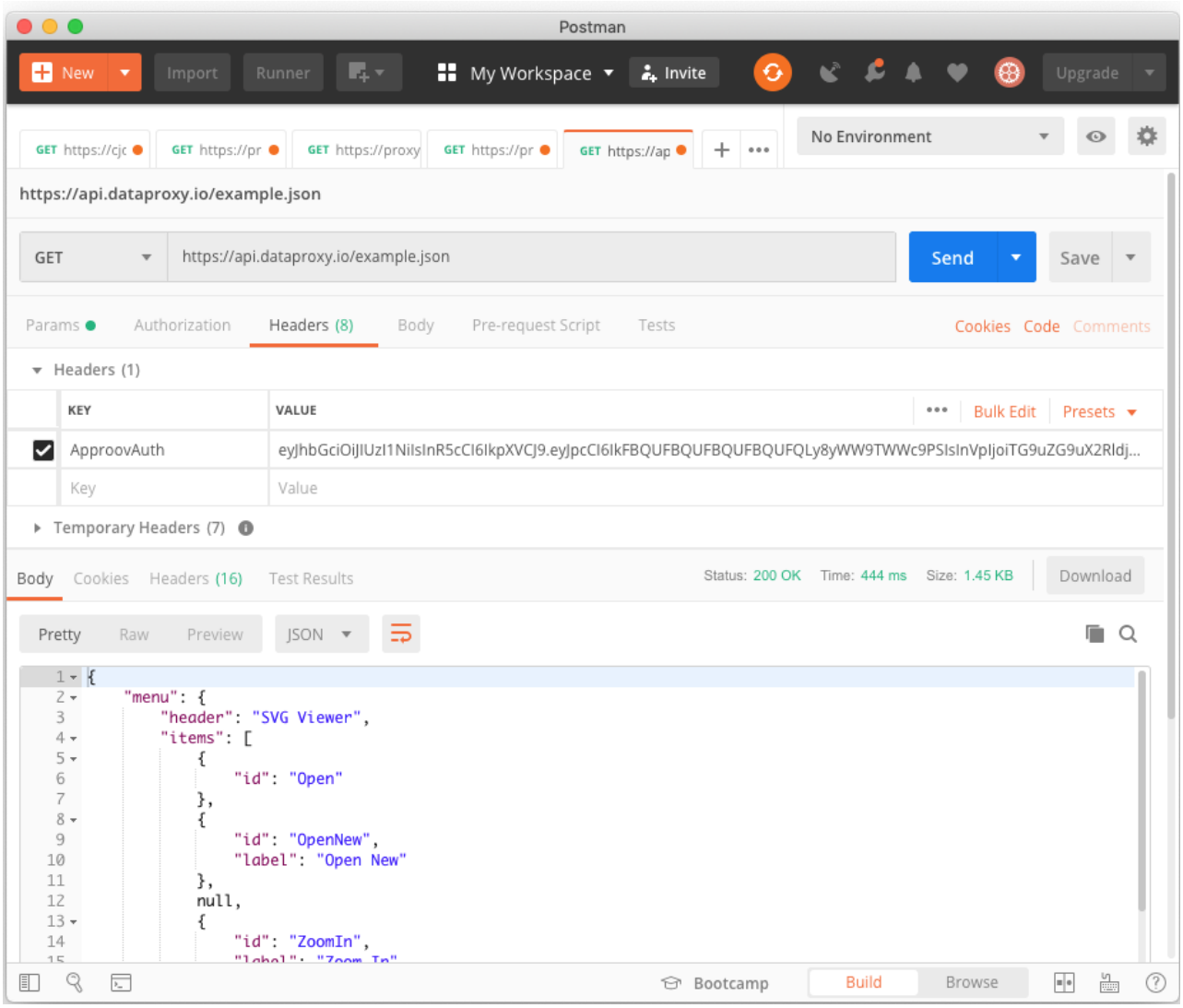
This will return an HTTP status code 401 with the message body of message:unauthorised



If you have noticed the time for the transaction is 103ms, this is because API gateway has returned a 401 immediately as there is no Approov-Token header and value.

But If we have a valid token, using the long dated token we have just generated in the previous section, then the return will be a 200, as the below shows.

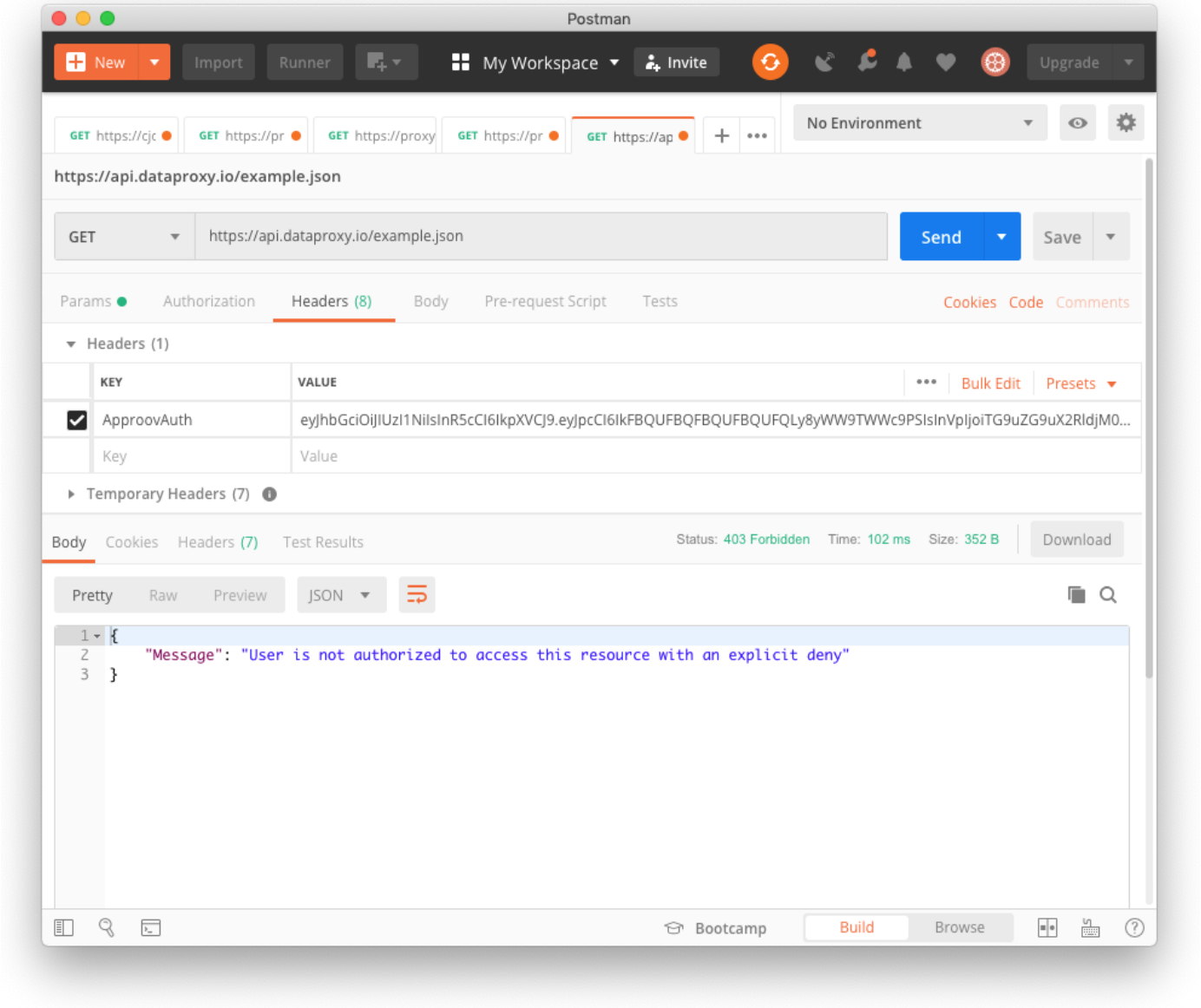
With this example I do not have the data cache enabled, therefore its transaction time is 740ms.



If we enable the data and token cache, it will return a better transaction time of 131ms, verses 740ms, a five fold increase in transaction time performance.

Now change a single character within the token, invalidating the signature.

You will now get a 403 return. An Explicit Deny, means, that due to the Token being corrupted and therefore invalid, we will create a deny policy for this token.



With Postman you can test different authorisation types, add path and query parameters to your request, and configure tests, it has quite a number of features that you can use, all for free, if you stay within the traffic and other licence restrictions.

CURL

curl is a fully featured command line application.

Available for Linux, Unix, Windows and macOS.

You can install it on centos/linux with “yum install curl”, or “apt-get install curl” on Ubuntu.

However, it is generally part of the operating release for cloud based servers.

With curl its nice and simple, for all unix/linux versions:

Export the token into an environment variable (for static ips this can really be part of your profile)

Then run the command, with “-H <header>:<token>” :

```
# export token=eyJhbGciOiJIUzI1NiIsInY7cCI6IkpXVCJ9.eyJleHAiOjE4Nzk1OTgwMjgsImlzcyI6Ik1vbml0b3IifQ.rTNTZfkRTuF-kak3zi7eNKlC09qrTMBs3ywIupu-TPo
#
# curl -H "Approov-Token:${token}" https://api.dataproxy.io:/example.json
{"menu": {
  "header": "SVG Viewer",
  "items": [
    {"id": "Open"},
    {"id": "OpenNew", "label": "Open New"},
    null,
    {"id": "ZoomIn", "label": "Zoom In"},
    {"id": "ZoomOut", "label": "Zoom Out"},
    {"id": "OriginalView", "label": "Original View"},
    null,
    {"id": "Quality"},
    {"id": "Pause"},
    {"id": "Mute"},
    null,
    {"id": "Find", "label": "Find..."},
    {"id": "FindAgain", "label": "Find Again"},
    {"id": "Copy"},
    {"id": "CopyAgain", "label": "Copy Again"},
    {"id": "CopySVG", "label": "Copy SVG"},
    {"id": "ViewSVG", "label": "View SVG"},
    {"id": "ViewSource", "label": "View Source"},
    {"id": "SaveAs", "label": "Save As"},
    null,
    {"id": "Help"},
    {"id": "About", "label": "About Adobe CVG Viewer..."}
  ]
```

There are hundreds of options and command line settings for curl.

COST COMPARISONS

CLOUD PROVIDERS

The following comparisons are for an API whose clients average 15 million requests per month.

However all cloud providers provide different types of API services, for example, Google uses APIGEE with a high starting price, but it is a more fully featured system with built in features, it's a more “out-of-a-box” service but comes at a much higher cost that Azure or AWS Services.

Azure is more similar to AWS, The Azure API Management service (launched in september 2018) is for interfacing Azure Serverless systems, But the pricing is more fixed.

Disclaimer: The following prices and costs calculate below were estimated for this particular example of an AWS API Proxy, at the time of writing. Prices and costs for various AWS services, products and solutions are constantly changing. You will need to complete your own cost structure for your particular environment to be able to successfully estimate the cost of your solution.

AWS

The AWS system we previously built, we used a lambda function (authoriser) , API gateway, and caches for the token and data.

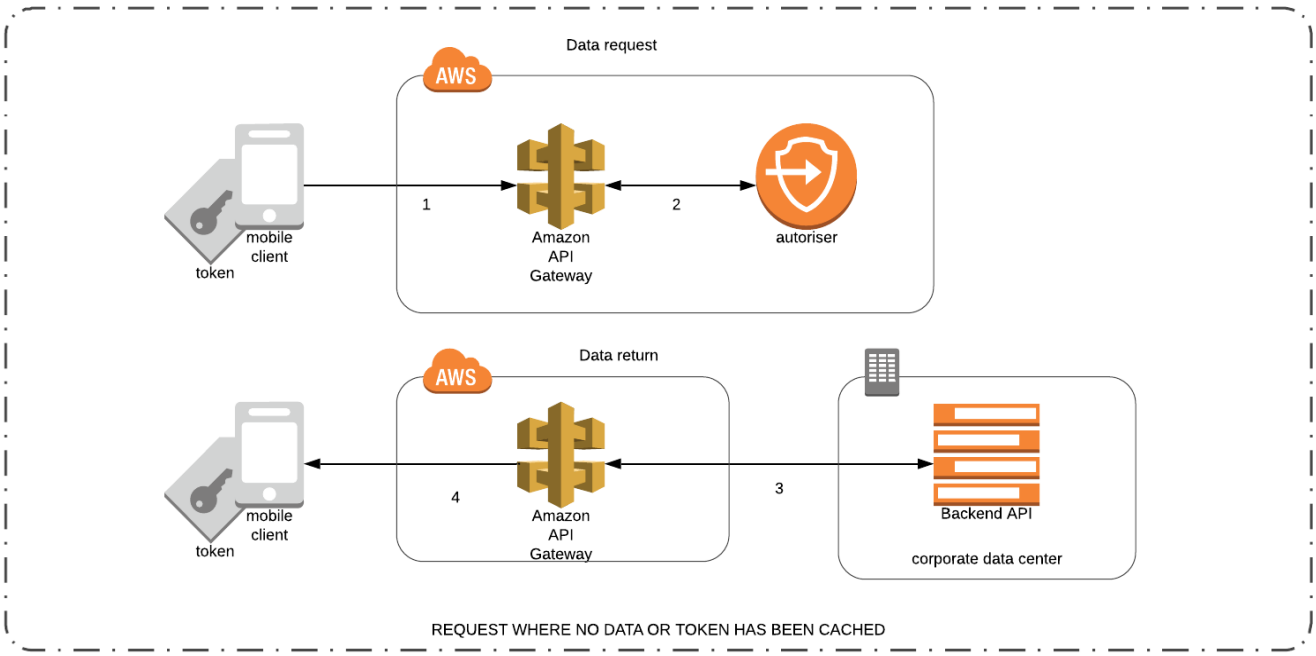
AWS charges only for what you use, incoming requests, data transfer, and cache use.

API gateway requests are $3.5 per million requests, ongoing data is charged at $0.09 a GB, incoming data is free.

API Gateway is capable of 10,000 per second, by default this rate is immediately burstable to 5000.

The 10,000 max rate is a soft limit, this can be increased by AWS if required.

The data flow and costs for no caching in the API are as follows:



If no token or data caches are enabled, the API will act as a straight through proxy with authentication provided by an Approov token validator (authoriser).

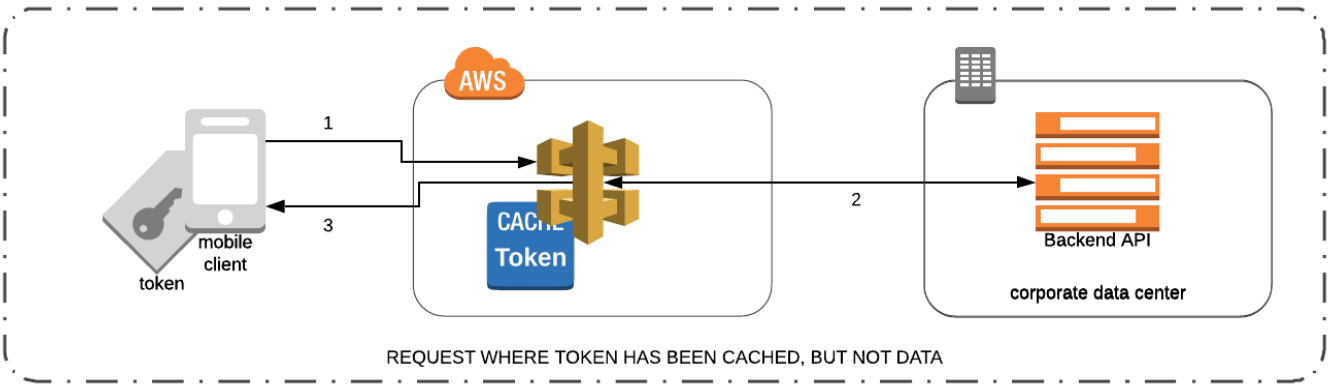
Assuming a 256 byte payload (token) and a 1Kb payload data return

1 (request@0.0000035) + 2 (100ms/128MB@0.000000208 + data@0.00000001) + 3 (data out(0.0000000225) + data in (0 charge) + 4 (data@0.00000009)

Per request = 0.000003898

Per million requests = $3.898

Data flow and expected costs for an API that has no cached data but caches token conditions is as follows:



API Gateways data cache is only used to cache GET requests, POST requests go directly through to the origin, I have included example pricing for posts and gets. There isn't a get deal of difference, as only outgoing data is charged for, data into AWS is rated at zero charge.

Assuming a 256 byte payload (token) and a 1Kb payload data return

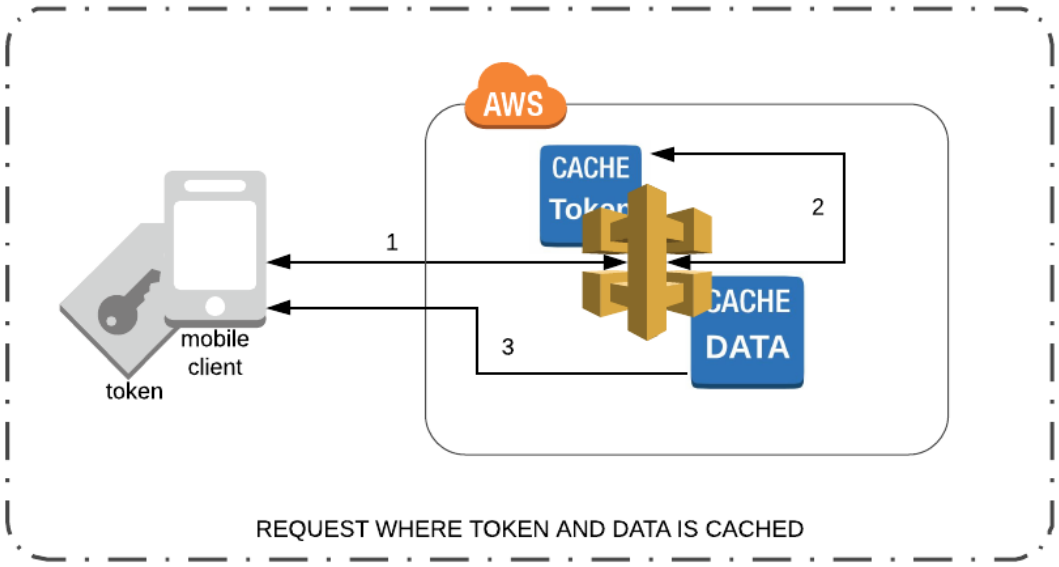
1 (request@0.0000035) + 2 (data GET out(0.0000000225) + data in (0 charge) + 3 (data@0.00000009)

Per request = 0.0000036125 GET

Per million requests = 0.0000037025 POST 1Kb data + 1Kb data return

Million requests = $3.61 GET transactions

Million requests = $3.70 POST transactions.



Assuming a 256 byte payload (token) and a 1Kb payload data return

11 (request@0.0000035) + 12 (0.0) + 13 (data@0.00000009)

Per request = 0.00000359

Per million requests = 3.59

Fixed Costs per API:

Fixed costs = Cache per 0.5GB = $14.88 month (small number of endpoints ~ 10, and tokens ~ 2000)

Take multiples of minimum depending on hit rate.

Cache TTL hit rate :

- 60 seconds TTL for rapid changing data, hit/miss rate 60/40 (So Cache handles 60% of requests)
- 300 TTL for medium, hit/miss rate approximately 85/15 (so Cache handles 85% or requests)

NOTE: you can invalidate entries, as well as the entire cache the entire cache.

AWS Total cost including cache is approximately $58 per month for 15 million requests, without any caches enabled.

GOOGLE

$ Cost comparison:

Google Apigee (excluding data cost) @15million requests = **$500 a month**

and you still have configure and code the apigee API.

More an out of the box solution, however, Apigee also comes with built in analytics.

AZURE

Azure brought out its Azure API Management last quarter 2018.

Pricing is based on per hour, for an API capable of handling a certain number of requests per hour.

15 million requests per month is, approximately, close to 6 per second based on a 24x31 average per month.

Not a high rate.

We will calculate using their Developer version which has a max request rate of 500 per second, the developer version has a fixed price of $0.07 per hour

There is no charge for outgoing data.

However the developer version only comes with a 50MB cache, a 1/10th of AWS Gateway's minimum cache size, there is no SLA or even an uptime guarantee.

So 15 million requests, with a 1KB payload works out at $0.07 x24x31 = **52.08 per month**.

Azure's API can be interfaced with Azure services or external on-premise origins.

FURTHER ENHANCEMENTS

CONTENT DELIVERY NETWORKS

You can front face your API with a CDN that is capable of Validating Approov Tokens

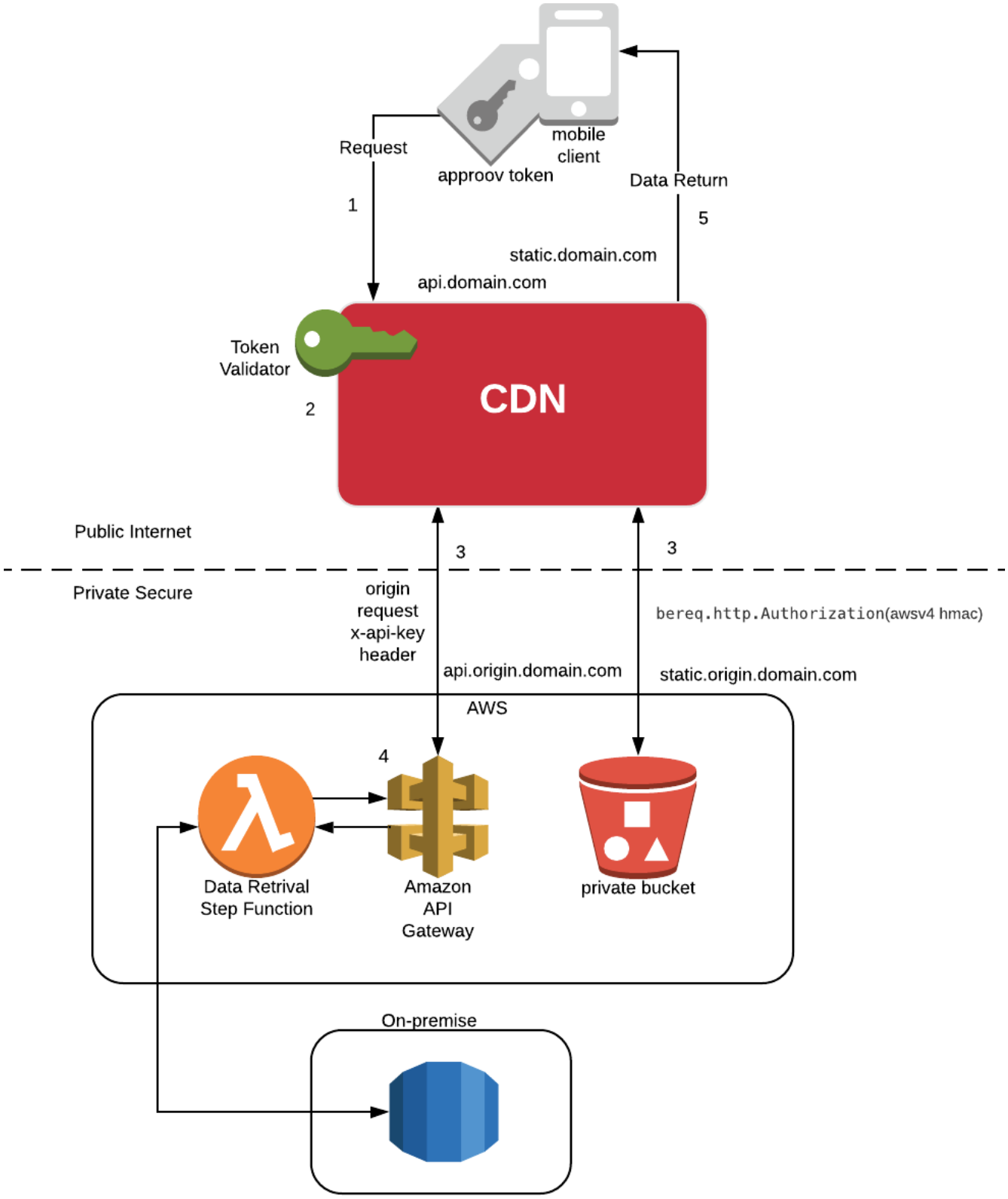
CDNs like AWS Cloudfront and Fastly as well as a number of others are capable of handling validations.

CDNs need to validate every request for data, if the Time-To-Live (TTL) is set greater than 0, then data inside the CDN cache will be immediately returned to the calling client after the requestor has presented a token and this token has been successfully validated.

The origin can be API gateway or other resources like S3 for access to static content.

This means that all data behind the CDN can be protected by Approov.

An example implementation is as follows:



1 Request for API data (static files or dynamic data) is sent to the domain hosted on the CDN, with an Approov generated JWT as the authentication/authorisation token.

2 Token validator within the cdn vcl checks the validity of the token (request ip matches claim ip and expiry timestamp is valid).

If the data is in the cache, data is returned to the client immediately.(**5**)

If the TTL has reached 0, it has expired, (we are assuming data is invalidated/or timeout, and not manually invalidated like write-through cache entry managed via a cache manager), then in the case of an API data request, the CDN will call the origin with the CDNs embedded x-api-key, for the data refresh.(**3**)

Some CDN's are capable of returning stale data to the client whilst the request for data refresh from the CDN is successfully returned from the origin.

4 Origin request goes through lambda step function (assuming cdn requestors API key is valid), the step function triggers a call to the on-premise database, gets the data, and returns data back to CDN API, who returns the data to the validated client.(**5**)

In the case of a static data request, (**3**) where the file required is kept on a private S3 bucket, the CDN will use inbuilt, long term, AWS credentials and make a signed request for the file located inside the bucket. S3 checks credentials used within the request (signed via AWS SDK and token) with IAM (not shown), IAM returns with an allow policy on objects within the bucket, data is returned to the CDN, via the request.

For API gateway requests from frontend CDN to origin, it is a simple API key, for access to S3 and other AWS services, the CDN will use long term credentials (AWS STS generated), made up of a key, a secret, a token) the CDN will then call S3 with a signed request for access to the bucket, and return the static data to the client and store the data within the CDN using the CDNs default TTL (86400).

EXAMPLE CODE

Git repository, with:

- Lambda Authoriser function
- Generator and validator.