**\_\_\_\_\_\_\_\_\_\_\_API GATEWAY (ESSENTIALSS)\_\_\_\_\_\_\_\_\_**

--------------------------------------------------------------------------------------------

***Purpose****: As part of the serverless architecture research here I will include interesting and relevant topics and features involving Lamba, API gateway, SQS, Dynamo, Ec2 and some other services that can provide an interesting feature within this type of architecture.*

*Note that all the following concepts are based on the following Udemy course and personal research, it is always a good idea to check for changes or updates from AWS to the lambda or other services used during this guide.*

<https://www.udemy.com/course/aws-lambda-serverless-architecture/>

The concept is simple, a full managed API within AWS in which you can define resources and methods like get, post etc. You can also link the API with different backends services or your own defined backend within Ec2, the case of interest for us is integrating API gateway with Lambda as the backend.

***Note: Usually this kind of infrastructure is not handled through the console of AWS, it is considered a good practice to use CloudFormation, SAM or other frameworks that allows you to deploy infrastructure using templates. I’m going to do a corresponding demo on different features using CloudFormation and SAM.***

***Keep in mind that SAM is just an abstraction of CloudFormation that is more oriented towards serverless design patterns but anything that can be done within SAM can be done within CFN and vice versa (there are some features within SAM like local testing and easy CLI deployments that can be useful so I will use SAM but inside of SAM I will also be using CFN syntax which is very similar)***

The API gateway service offers a lot of useful features like request validation and authorization as well as versioning and canary deployments which allow you divide your traffic within different stages of your API.

***Note: Aside from CFN and SAM there are third party tool providers like the serverless framework, some of the advantages of that framework is that it is designed to work with a multi-cloud architecture, and it has a community that develops plugins for it with extra functionality, one drawback is that the new features that AWS implement could be late for that framework.***

Concepts  **//**

There are different types of APIs we can create with this service:

**HTTP API:**

It offers less features than a rest API, but they are generally offered at a lower price and with a minimal technical overhead.

**REST API:**

Choose REST APIs if you need features such as API keys, per-client throttling, request validation, AWS WAF integration, or private API endpoints.

**Note: Find which one suits you better here** [**https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-vs-rest.html**](https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-vs-rest.html)

**Socket API:**

API Gateway WebSocket APIs are bidirectional. A client can send messages to a service, and services can independently send messages to clients. This bidirectional behavior enables richer client/service interactions because services can push data to clients without requiring clients to make an explicit request. WebSocket APIs are often used in real-time applications such as chat applications, collaboration platforms, multiplayer games, and financial trading platforms.

(Is just an open connection)

<https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-websocket-api.html4>

**Private Rest API:**

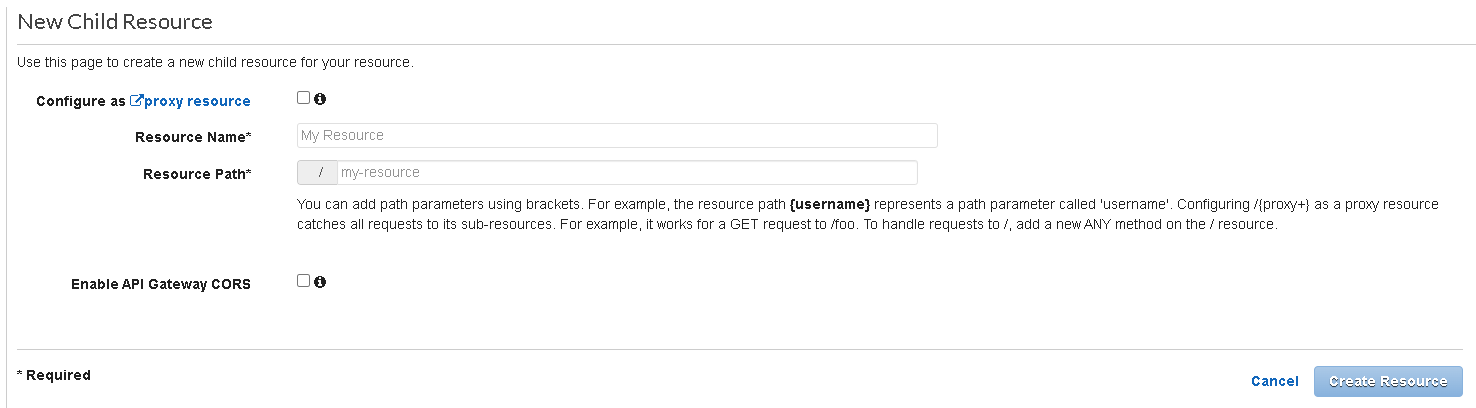
In general, is the same as a REST API but it is only accessible from within a VPC.

**Personal Note: To this research the Rest API seems to be the more suitable.**

In API gateway we have the concept of resources which is nothing more than the path for an endpoint, an example of a resource could be /users or /fetch-Data, each of those resources can be mapped to one or more http methods like GET,POST,DELETE and so on.

An API can be deployed to different stages, a stage is just like an environment, so with just one API we can run that same API within a test environment or a prod environment or custom environments which can al be versioned or integrated with different environment variables.

It offers fine grained access control as well (I will explain this later)



The resource path parameters are specified within curly braces like this /user/{id} = /user/1 this will then be accessible from our lambda function.

INTEGRATION TYPES //

When creating a HTTP method for a resource we can specify different integration methods.

!!! THE FOLLOWING PART COULD BE REALLY IMPORTANT I SUGGEST TO PUT THE EFFOR TO DIVE DEEPER IN THE FOLLOWING CONCEPTS !!!!

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**Lambda function:**

It means that this method will be integrated/linked to a lambda function, in this case we must specify the region of the lambda function and the name, ARN or alias/version.

We can also select a timeout for the http method which is by default 29000 milliseconds which is 29 seconds.

**Note: Keep in mind that all this configuration is per method per resource, that means we can use lambda functions which are spread over different regions, and we can also use different integration types within our API.**

**HTTP:**

This allows you to connect your API with an HTTP endpoint, so like in VPC link now the API gateway API acts as a client and can call endpoint from the web or another API you have running in an EC2 instance for example with express.js.

**VPC link:**

This is an integration type that allows you to connect your API to different resources within a VPC in a secure way without exposing them to the public internet. It is like a two steps process in which a client calls API gateway and then API gateway acting now as a client calls the VPC endpoint.

**AWS service:**

Whit this type of integration the request is passed from API gateway towards a service you specify so a client could invoke a method from DynamoDB through our API gateway API for example.

**Mock:**

With this integration type there is no other service as a backend the response is purely handled by API gateway on itself using mappings and transformations (this features will be discussed later on).

**Note: This could be useful for testing, very simple workflows that doesn’t require a full lambda execution or request correction validation.**

**I highly recommend reading this documentation since there are a lot of useful features within the different integration types:**

[**https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-api-integration-types.html**](https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-api-integration-types.html)

Non proxy integration vs proxy integration:

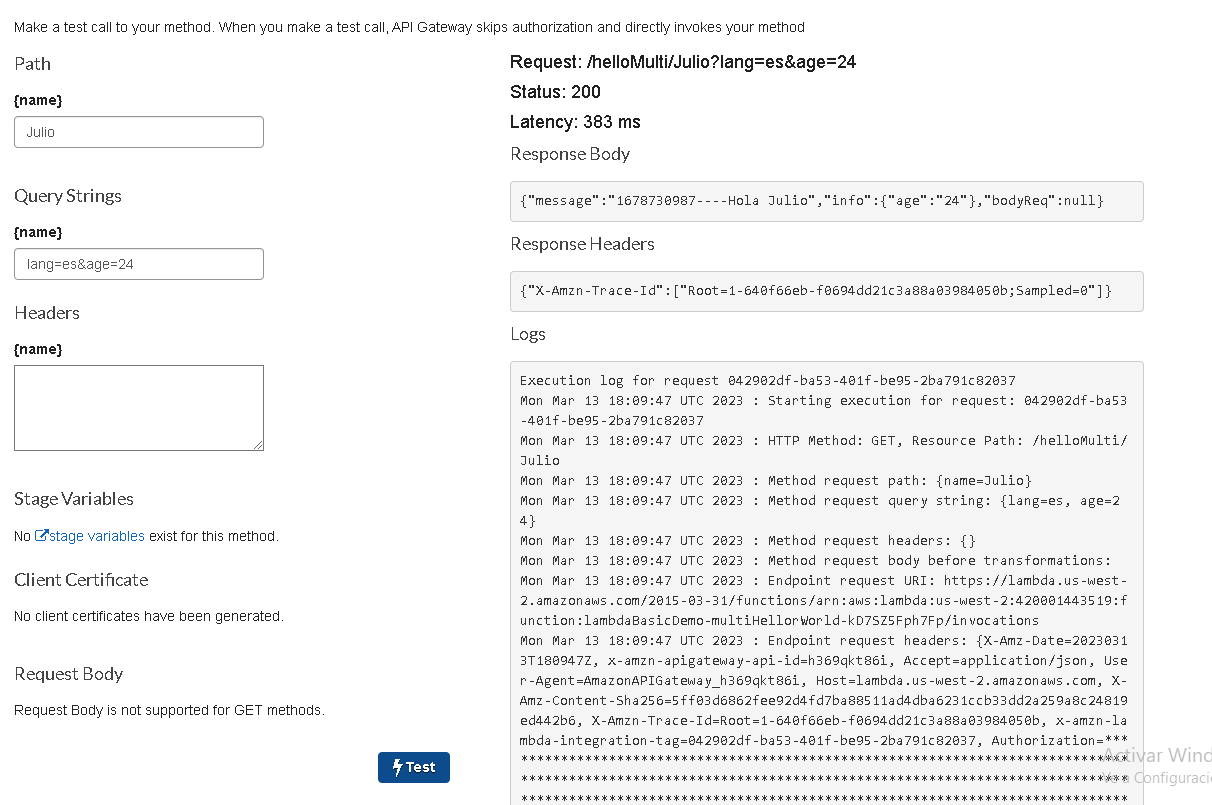
In a proxy integration API gateway receives all the HTTP request from the client and forwards it as it is to the backend service and similarly API gateway receives the response from the service and send it to the client as it is.

In a non-proxy integration API gateway allows you to receive and return only a specified part of the incoming request and the response, so here you can modify how the request reaches the backend and how the response reaches the client, to achieve this API gateway uses mapping templates and request/response transformations.

Is important to differentiate these two values since depending on the integration type you choose you can use one or the other, or both (in the link above you can see which proxy integration type does is integration type support)

**Note: API gateway needs to have permissions to invoke the lambda functions, more specifically the resource and method that are trying to invoke the lambda function.**

One of the useful features the API gateway console offers, is being able to test your endpoints/resources without having to use postman or other third-party tools, which are still useful to test that the deployment was successful and that it can be reached from anywhere but in the test overview you can test for debug purposes before deploying in a faster and simpler way.



Before any changes can be accessible from the internet or the overall main workflow of the API you need to deploy the API to a stage, for example deploying your changes to a development stage, then a URL will be generated, and you can use it to call you API endpoint from postman for example. Also keep in mind that changes on backend services doesn’t require a redeployment from API gateway since they are isolated, only changes within API gateway requires a redeployment, CFN usually and practically always detects when a resource needs to be updated/re-deployed.

CORS  **//**

CORS stands for cross origin resource sharing, and it is just a mechanism that works with the headers in which the server indicates any origin other than its own (that’s running from a web browser script) that should be allowed to use the resources (third party applications for example ex: our app consulting Google’s API)

We can test CORS with the following website: test-cors.org

The header used for this mechanism is called Access-Control-Allow-Origin, this header must be included within the response so the access can be granted.

In case of using **lambda proxy integration** (or any proxy integration method) this header should be sent directly from the backend service, in this case the lambda function.

Keep in mind that most moder browsers sends what is called a preflight request or options request before the actual real HTTP request and only if the preflight request is successful then the browser will send the original HTTP request.

To send header from a lambda function we can return an object that looks like this:

return {

statusCode:200,

headers: {

“Access-Control-Allow-Origin”:”\*”,

},

body: JSON.stringify(response)

}

This header will tell API gateway that this endpoint can be accessible from all the website or all the domains.

When we enable CORS for a resource an OPTIONS method is created, this options method will run before the real HTTP request, and it is in charge of returning the Response headers which will contain the Access-Control-Allow-Origin and other useful headers.

**Note: You can read more about CORS in API gateway in the following link:**

[**https://docs.aws.amazon.com/apigateway/latest/developerguide/how-to-cors.html**](https://docs.aws.amazon.com/apigateway/latest/developerguide/how-to-cors.html)

Request validators //

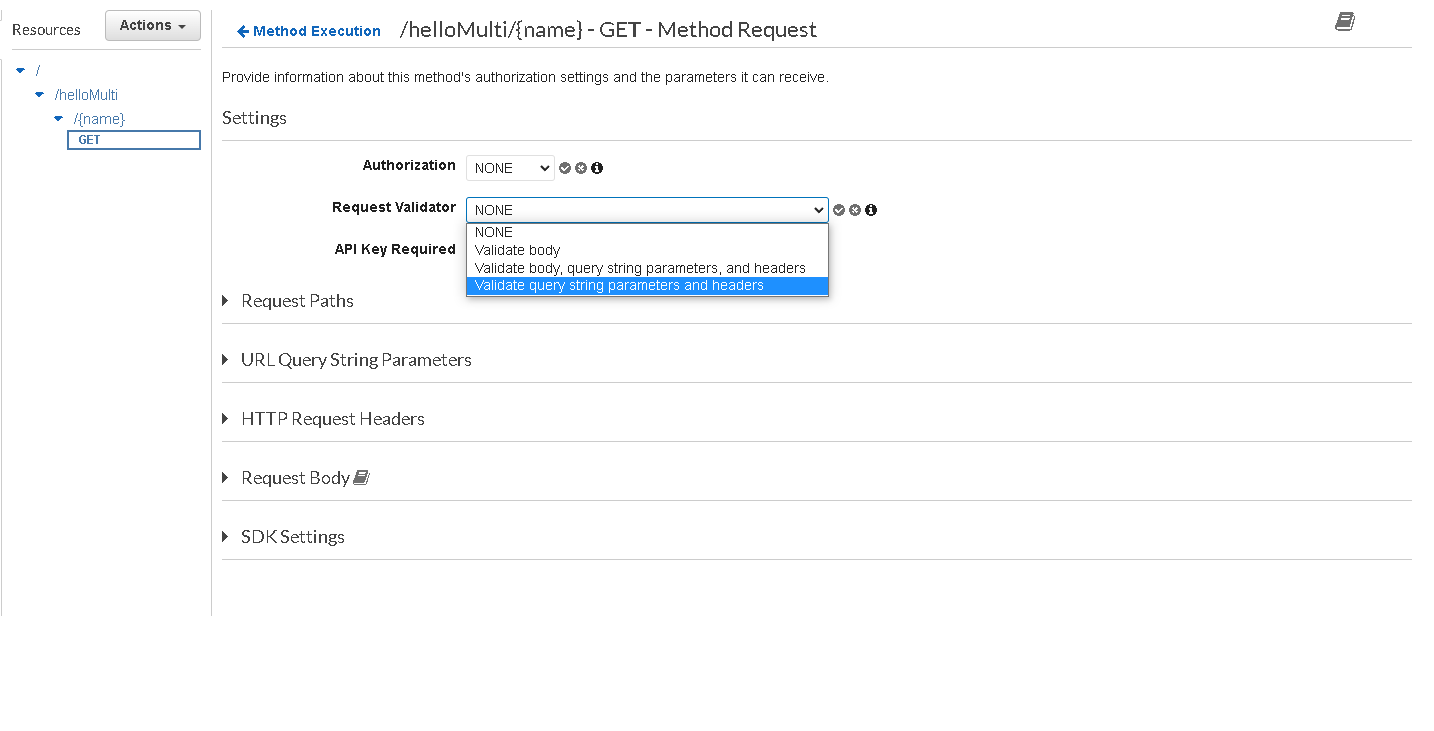
There are 4 blocks of interaction within API gateway that can happen, Method Request which happens before the request reaches the backend, integration request which happens after method request but also before reaching the backend (note that the integration request is only available if we are using non-proxy-integration), then there is the integration response that happens before delivering the response to the client (note that the integration response is only available if we are using non-proxy-integration) and lastly there is the method response which is the las step before the response reaches the client.

The ideal block to validate the request is within the Method request block.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Within the Method request we can validate the body, body along with query string parameters and headers or query string parameters along with headers.



Note that in this part we can control access in different ways like using API keys or different authorization methods, I will detail them later.

The way it works is that we specify header that MUST be present within the request, otherwise that request won’t be process further, it works in the same way for query string parameters, here we can also cache those values.

**This can be very useful specially in the case that we use lambda as the backend since this will prevent a lot of wrong executions and invocations of the lambda functions.**

**Note: remember than when we use non lambda proxy integration the event parameter within lambda will be a custom event so it will look as we define as opposed to the test events examples that we saw previously that were generated for example with s3 or the API gateway proxy integration.**

Mapping and Transformation //

When we use non proxy integration there might be a case in which we need the request to be formatted in a specific way, for example, our lambda function specs an object with two keys but the client made the request with one key and the other value was sent within the URL, we can transform the request to get that value from the URL and send it to lambda as it is spected.

All this procedure can be done within the Integration Request block using mapping templates.

Within the integration requests and option can be specified called Mapping templates where you can select 3 types of workflows:

You could select the **"When there are no templates defined (recommended)"** option in the API Gateway integration settings. This would ensure that the incoming request body is passed through to the Lambda function without any mapping or transformation being applied.

You could select **the "When no template matches the request Content-Type header"** option in the API Gateway integration settings. If a request comes in with a Content-Type that doesn't match any of the defined templates, you could select the "Passthrough" option to ensure that the request body is still forwarded to the backend service without any transformation.

The **"Never"** option in API Gateway integration settings means that the request body will never be passed through to the backend service. This is useful if you don't want to forward the request body to the backend service for security or performance reasons, or if the backend service doesn't require the request body to process the request.

Now back to mapping templates, we can create a mapping template in a to map a variety of content-types, most used content-type case it application/json.

We can select **models** (I will talk about them later) or any predefined template we have created before and of course we can create the custom template.

Example:

Let’s suppose we have a lambda function that is expecting the following object structure:  
{

“operation”:”add”,

“input”: {

“operand1”:5,

“operand2”:4

}

}

But the request doesn’t arrive with a body like that, instead the operation comes as a query String parameter and the body just looks something like this:  
{

“operand1”:5,

“operand2”:4

}

Using the VTL syntax (Apache Velocity Template language) we can map and transform the request to create a body that will be expected by the lambda/backend.

**Note: you can read more about mappings and VTL syntax here:**

<https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-mapping-template-reference.html>

<https://velocity.apache.org/>

It essentially works because it lets us access some variables within a JSON defined object for example writing this $input.body will give us access to the raw body of the request and we can use it wherever we want within our template, in the link about you can see the usage of those variables.

To access the query string parameters for example we could use $input.params()

The mapping template would end looking something like this:

{

“operation”:”$input.params(‘operation’)”,

“input”: {

“operand1”:$input.json(‘$.operand1’),

“operand2”:$input.json(‘$.operand2’)

}

}

**Note: The $ symbol represents the JSON root of the incoming request body.**

**There are other powerful tools within the VTL language like if statements, loops, regex, etc. However, keep in mind that the use of those features should be very well implemented and is not recommended or very good/common practice to do it, since it is recommended to not split logic process between API gateway and the backend.**

**One scenario where this could be useful is if you are using the mock integration type.**

It’s important to note that this whole process can be also done within the response mapping to modify the response from lambda/backend before it reaches the client. This template can be assigned for each response code, for example you can specify which template to apply if the status code was 200 or if it was 400.

Preflight and CORS details //

When a cross-origin request is made, the browser sends a preflight request (OPTIONS method) to the server to check if the actual request (GET, POST, etc.) is allowed. The preflight request includes the Access-Control-Request-Headers header that lists the headers that will be sent with the actual request.  
The server (your API in this case) receives the preflight request and checks if the requested origin is allowed to make the request. If allowed, the server responds with the Access-Control-Allow-Origin header that specifies which domains are allowed to access the resource. The server may also include other headers such as Access-Control-Allow-Methods and Access-Control-Allow-Headers that specify the allowed HTTP methods and headers, respectively.  
Once the browser receives the response from the server, it checks if the origin is allowed to access the resource. If allowed, the browser sends the actual request (GET, POST, etc.) with the appropriate headers. If not allowed, the browser will not send the actual request and an error will be thrown.

Keep in mind that not all CORS requests require a preflight request. Simple CORS requests, which meet certain conditions, do not require a preflight request.  
A simple CORS request meets all the following conditions:

1. Uses one of the following methods: GET, HEAD, POST
2. Uses only safe header fields. Safe header fields are those that the CORS specification defines as simple headers, including:

Accept

Accept-Language

Content-Language

Content-Type (but with a limited set of values)

Last-Event-ID

Content-Encoding

1. If the request includes any headers that are not safe, or if the Content-Type header has a value other than application/x-www-form-urlencoded, multipart/form-data, or text/plain, then it is not a simple CORS request, and a preflight request is required.
2. Does not use the XMLHttpRequest upload event listeners.

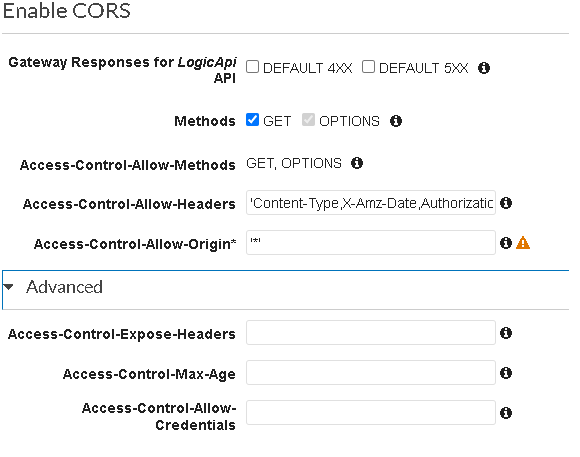
If a request meets all these conditions, then it is considered a simple CORS request and the browser will not send a preflight request. Instead, the browser will include an Origin header in the request and the server will respond with the appropriate Access-Control-Allow-Origin header to allow or deny access to the resource.

When using non-proxy-integration, to correctly set the CORS header we need to modify the options method response from our method to send the header Access-Control-Allow-Origin,

It is as easy as just writing the desired header and map its value within the method response block.

Note that we can specify the headers we want to return with the response in the method response block but when we want to map them, we need to use the integration response block, there we can assig a mapping value to the headers depending on the status code so we can map different values depending on what the lambda function returned. In summary you create a header within method response, then this header will automatically appear in the integration response for all status code, and you can then individually specify a value for it.

Optionally in the console while selecting our resource there is a button that says “enable CORS” that essentially does the same and even provides a more advance configuration for different types of headers involve with the CORS mechanism like Access-Control-Allow-Credentials.



**Access-Control-Allow-Methods:** This header is used to tell the browser which methods should be allowed within a request for a specific resource.

**Access-Control-Allow-Origin:** This header is used to tell the browser which domains can make request to our API.

**Access-Control-Allow-Headers:** This header is used to tell the browser which http headers are allowed within a cross origin request from a browser.

**Access-Control-Expose-Headers:** This tells the browser whether a custom header within the response is accessible from the client using JavaScript.

**Access-Control-Max-Age:** This header indicates how long a preflight request response should be cached by the browser, so, how long until a new preflight request has to be send again.

**Access-Control-Allow-Credentials:** Indicates if a cross origin request can include credentials like cookies, HTTP authentication, by default the requests can’t have credentials (keep in mind that this could cause security risks if not used correctly)

Models and JSON schemas //

The following concepts could be very useful I suggest examining this section with all the detail.

You can find more information about JSON schemas here: <https://json-schema.org/>

The models provide a JSON schema for the mapping templates used for the request and response transformations; they can also be used for **request body validation** (note that the last one can be used even if lambda if configure with proxy integration)

**Models For Body Validation:**

API gateway by default has to predefined models called Empty and Error but we can also create our custom models.

We can select a model name, the content-type that is validation and write a model description, if we want to validate an incoming body request that MUST look like this:  
{

“operand1”:5,

“operand2”:4

}

We can create a model like this (Keep in mind that the request will get processed only if the body exactly match our model):

-We define each or the attributes expected on the request body.

-The **$schema** key is mandatory.

-We define the type of the root attribute of the body which in this case would be an object since the request was made with application/json content-type.

-Then we define the properties spected within the object property.

{

"$schema": "https://json-schema.org/draft/2020-12/schema",

“title”: “body validation model”,

“type”:”object”,

“properties”: {

“operand1”:{“type”:”number”},

“operand2”: {“type”:”number”}

},

“required”: [“operand1”,”operand2”]

}

Now that the model is ready it can be used within the method request block selecting any request validator option which includes body validation, then we specify the model we want to use within the request body section and the content type we want to validate.

Note that when we create a model it can if we want to be applied across multiple resources and methods so it is available within all of our API resources while only writing it once.

**Models For Body mapping (only for non-proxy integration):**

The main use case for this is if we want to use a mapping template in multiple API resources, so, instead of copying and pasting the same template again we can create a model which exactly matches the structure of the mapping template and reuse it whenever we want.

This was the mapping template I talked about previously:

{

“operation”:”$input.params(‘operation’)”,

“input”: {

“operand1”:$input.json(‘$.operand1’),

“operand2”:$input.json(‘$.operand2’)

}

}

And this would be the corresponding model for this template that correspond with this json structure:

{

"$schema": "https://json-schema.org/draft/2020-12/schema",

“title”: “Test mapping model”,

“type”:”object”,

“properties”: {

“operation”: {“type”:”string”},

“input”: {

“type”:”object”,

“properties”: {

“operand1”:{“type”:”number”},

“operand2”: {“type”:”number”}

}

} ,

“required”: [“operand1”,”operand2”]

},

“required”: [“operation”,”input”]

}

**Note: overall, the Models are just a JSON that indicates how the structure of a defined content type should look like.**

Now when we select this model within the mapping templates, we will get something like this which is just a placeholder so we can just define from where should our template find the corresponding values.

**Remember to review the VTL language syntax documentation.**

#set($inputRoot = $input.path(‘$’))

{

“operation”:”foo”

“input”: {

“operand1”:4,

“operand2”:6

}

}

Then we can replace the placeholders like this:

#set($inputRoot = $input.path(‘$’))

{

“operation”:$input.params(‘operation’)

“input”: {

“operand1”: $inputRoot.operand1,

“operand2”: $inputRoot.operand2

}

}

Error Handling (Gateway responses) //

The Gateway responses are simply the HTTP error responses that gets triggered within API gateway while processing a request (note that this is isolated from the error codes our lambda/backend could return)

This error responses can be customizable, there are some predefined error responses gateway could throw, there are also default responses for 4XX or 5XX and of course we can create our own error responses.

Series 4XX indicates a client error, so, something that the client can fix to API gateway to work properly while 5XX series indicates an error in the “server” side.

Inside each response code we can configure headers that should be send along with the error response, we can specify the content type and also use mapping templates to generate the response body (in this case the context object/variable available within the VTL language can be very useful)

Note that if there is nothing specified for errors like 403 Access Denied then the settings used for the default series 4XX will be used, the same behavior would happen for series 5XX

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Post deploys options Logging and Monitoring //

Remember that every time we deploy our API it must be deployed to a specified stage once that’s done some tunning settings and other useful features will be presented to us, which includes not only post configuration and options for the API stage in general but also some of them are available within each resource and method so it has granular configuration, some of these features are the following:

**Cache settings:** It allows better performance and faster execution, keep in mind that this will result in an extra charge that’s not covered by the free trial, we can select the capacity of the cache to be within 0.5GB and 237GB, the cache data can also be encrypted, and we can specify the Cache time to live (TTL).

This cache configuration will cache the endpoints responses reducing the amount of calls made to the endpoint and improving latency.

When you enable caching for a stage, API Gateway caches responses from your endpoint for a specified time-to-live (TTL) period, in seconds. API Gateway then responds to the request by looking up the endpoint response from the cache instead of making a request to your endpoint. The default TTL value for API caching is 300 seconds. The maximum TTL value is 3600 seconds. TTL=0 means caching is disable.

There is a maximum size that a response can be cached (1048576 bytes) and remember that if encrypted the size could increase.

A useful example of this could be that a lambda execution could be saved if the response is cached since API gateway will look within the cache (if on TTL) and return the response from there, therefore, it is worthy to calculate pricing with this option in mind.

You can find all the detailed documentation about cache in API gateway for rest API’s here:

<https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-caching.html>

**Method Throttling:**

This option is used to protect the API and it’s resources from overload, we can set the rate limit to 100 request per second and once that limit is met then gateway will start returning 429 error (too many requests) in summary Rate limit is the maximum number of request that can be made within a certain period of time but there’s also a concept called burst limit which allows a short-term burst of traffic to exceed the rate limit before the throttling starts to kick in and then the rate limit will be applied like a brick wall.

In summary, the rate limit sets the average rate of requests allowed over time, while the burst rate allows for short-term bursts of traffic that exceed the rate limit, but only for a limited time. These settings can be used together to provide a balance between responsiveness and protection for your API.

Usually accounts have 10000 as the rate limit per second and 5000 as the burst limit.

**Client Certificate and Tags:**

Tags is a common term within AWS as it is just a way to “tag” different resources so they can be grouped and filtered within the console or programmatically.

We can assign a client SSL certificate to the API but this needs to be generated beforehand, the client certificates can be created very easily within API gateway, we can add a name and a description.

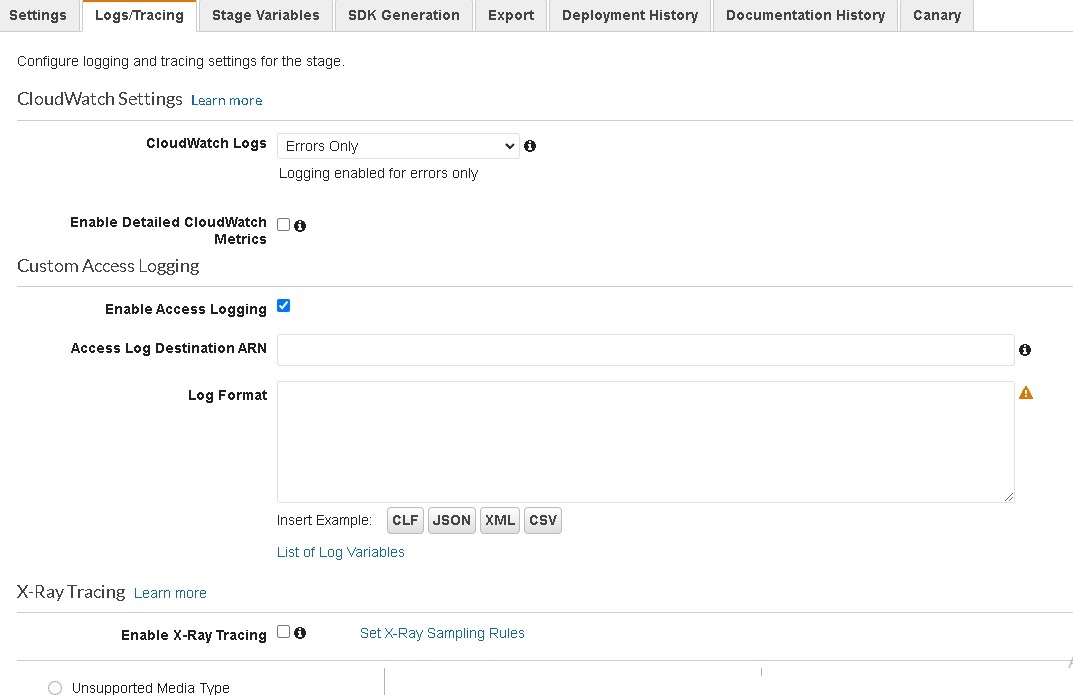
A client SSL certificate is a digital certificate used to authenticate clients that are making requests to your API in API Gateway. When a client SSL certificate is required for an API, the client must present a valid certificate to establish a secure HTTPS connection to the API Gateway.

**Note: By default, all methods inherit the API stage configuration however some of this configuration like cloud watch logs and throttling can be overwritten at the method level**

**Cloud Watch logs:**

**Note that logs are also chargeable however this is useful for debugging and benchmarks of the performance of our API.**

To enable the logs for the API/method remember to set the ARN of a role that has access to cloud Watch logs. The policy name for this is already provided by AWS and its name is: **AmazonApiGatewayPushToCloudWatchLogs.**



AWS X-Ray is a service that allows you to trace requests made to your application and provides insights into how your application is performing and where issues may be occurring. When you enable X-Ray tracing for an API Gateway stage, API Gateway integrates with X-Ray to capture tracing information about the requests made to your API and the responses returned.  
When X-Ray tracing is enabled for an API Gateway stage, API Gateway automatically adds tracing headers to incoming requests and propagates them through any integrated backend services. This allows X-Ray to generate a detailed trace of each request and provide insights into how long each component of the request took, where errors occurred, and where latency occurred.  
By enabling X-Ray tracing for your API Gateway stage, you can gain visibility into how your API is performing and identify any issues or bottlenecks that may be affecting performance. You can view the traces and insights provided by X-Ray in the X-Ray console, which includes information about the individual requests made to your API, as well as aggregated metrics and insights about the overall performance of your API.

Tabla

Descripción generada automáticamente con confianza baja

**Deployment History:**

It just shows a relatively small, detailed history of the deployments we had made, the useful thing about this feature is that it allows us to rollback to previous stages very easily.

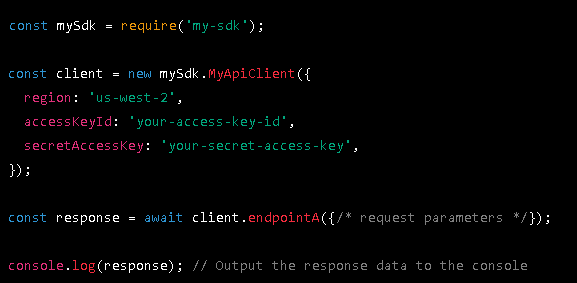
There are more useful features, but they will be covered in the deep dive part of the AWS API gateway guide.

SDK Generation //

We can generate a client SDK for different platforms/languages like JavaScript, Ruby, Java etc.

To use an SDK generated for your API Gateway, you typically install the SDK as a package in your application's codebase and then import it into your code. Once you have the SDK imported, you can call the API endpoints using the client methods provided by the SDK, as you described in your example.

We will get a zip file containing our generated SDK

This is useful so our App doesn’t have to write HTTP request code, it may end up looking something like this instead:  


Swagger files and Exporting the API definition //