**Note: For now, you can ignore the README.md file automatically generated with SAM CLI and the samconfig.toml file since those topics will be explained in a dedicated guide)**

Explanation //

**Purpose:**

In this demo we are going to create a new explicit API (which we will use to explore a lot of concepts from API gateway) with some endpoints with different integration types so we can review the weaknesses and strengths from each one, we will also include integration type of lambda function with non-proxy-integration and proxy integration, the proxy one will implement validators for the body, the headers and the query string parameters .

(Note that you can also specify path parameters that are mandatory looking for a specific value or a data type)

**Observations:**

I created this folder with name "implicit" meaning that this was going to use implicit resource creation that SAM provides, however now we can start to point some of the weaknesses of implicit resource creation such as:

-Limited control over the configuration of the implicit resources.

-Limited support for advance features of the resources

So, due to this, even if I called the folder with the word **Implicit,** we will use **Explicit resource definition** along with **implicit resource definition**, since we need some advanced features of API gateway, this resource will be defined explicitly but the lambda role for example will still get created implicitly although we could do it explicitly as well by changing the type from AWS::Serverless::Function for AWS::Lambda::Function or using the Role property (which is a must if we need our lambda specific access to different services) however keep in mind that if not necessary it can be better to stick to implicit resource creation since it reduces the general technical overhead.

The first thing I did was to define an **AWS::Serverless::Api** explicit resource and referenced the Api) in the function resource **(RestApiId: !Ref ApiGatewayBasics)** this way I can have far more control than with implicit resource API creation.

Note: Remember that logical names of the resources are like variables and can be referenced and we can even read its properties such as ARN, ID, etc. (you can find which properties are available for the different resources in the SAM resource documentation or in the CFN resource documentation) but keep in mind that it can be different from the REAL name of a resource that will be shown in the AWS console.

I used the swagger 2.0 api definition to create the API resource, if you don’t understand the syntax don’t worry since the swagger 2.0 and 3.0 API definition concepts and syntax will be covered within the guide/additional-guides folder.

Since swagger and AWS are two separated things there is a concept called SAM extensions which you can see more about here (they will let you extend the features of API gateway over swagger 3.0 api definition): <https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-swagger-extensions.html>

**Example:**

**To specify non-proxy for the resource I had to use the AWS extension “x-amazon-apigateway-integration”** setting its type to **'aws'** then in the Uri property I specified the following Uri:

**Fn::Sub: arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-31/functions/${calculatorFunction.Arn}/invocations**

That Uri is the endpoint of the backend for integration of the AWS service (in this case AWS lambda) so we are in fact having two clients in the whole process…we call API gateway and then API gateway calls Lambda, that’s why we specified the http Method twice, once within the path/function and another one inside the method definition for the specific resource integration type.

Notice how in the console it looks like lambda integration has its own non-proxy integration but that’s nothing more than a front end trick since in reality it is AWS integration pointing towards lambda service which visually offers a better UI/UX experience.

**IMPORTANT CONSIDERATIONS:**

During this demo I realized that local testing doesn’t work the intended way while using request validators, you can define the validators and deploy the stack first then test the validators within AWS API gateway and they will work fine.

*"When you run the API locally with sam local start-api, it uses the* ***local Lambda function and API Gateway emulator*** *to handle the requests, which do not enforce request validation by default. This is because the emulator is intended for local development and testing purposes, and the main goal is to provide a simple and fast way to test your API without relying on the cloud resources."*

Remember also that even if you can specify the request parameters like this within the function resource:

**- method.request.querystring.lang:**

**Required: true**

**Caching: false**

**- method.request.querystring.age:**

**Required: true**

**Caching: false**

It is not very well integrated, and it should be easier as well as better configurable to do it within the API resource. This is due to the RestApiId property within the lambda resource, if provided then the **RequestParameters** property will get ignore, resulting in this property only working for implicit API resource creation, you can see an example of this in the LambdaBasicsDemo demo where the API gets in fact created implicitly, and it’s good to know that when working with explicit resources some properties might get overwritten, unconsidered or show a poor/not-expected documented behavior.

Mock integration //

Now, moving on into mock integration type, I have included an endpoint called /**mockIntegration**, the main purpose of using the mock integration type is to test your API Gateway configuration and ensure that the request/response mapping templates are correct before deploying the API to a live backend.

With the mock integration type, you can simulate a backend endpoint without having a real backend. This allows you to test your API Gateway configuration and see how it would behave in a real environment without incurring any actual backend costs.

Additionally, you can use the mock integration type to generate sample responses for your API, which can be useful for documentation or for testing client applications that will be consuming your API.

***There are some tricks within this integration type like for example that the integration request mapping template accept all kind of “keys and values” but the only useful one is the status Code (unless you decide to use other key/values to define logic so you can define the statusCode within api gateway, remember that VTL language offer if ,loops and other useful tools) which will define what response integration template will be used, other interesting thing is that since mock integration doesn’t offer a backend the body is always lost within the integration request and is not reachable within the integration response, tho I found a trick (which I don’t suggest to use in relevant scenarios since this may get fixed/patched by AWS, the trick is to set the body to be a parameter within the integration request, since parameters like headers, paths and queryStringParameters are accessible in the integration response that will make the body accessible as well as transformable in the integration response.)***

***You can read about this workaround here:***

<https://stackoverflow.com/questions/47918477/aws-api-gateway-use-mock-integration-to-echo-response-body>

Note: This integration type could cause some very creative solutions. But keep in mind that it can’t be tested locally with SAM local it has to be deployed into a stack in CFN.

HTTP integration //

Now moving on to the http integration type we can actually chose if we want it to be a proxy or a non-proxy integration in which, the only major difference will be that API gateway will (or won’t) act as a proxy giving (or revoking) the access to apply mapping or transforms to the integration response.

The endpoint I created **(/httpIntegration**) has to methods, **GET,** this uses **http non proxy integration** and calls a test API over the internet (**https://jsonplaceholder.typicode.com/users**) and then returns its response without applying mapping or transformation but we could if we wanted to implement it since is a non-proxy integration, then we have a **POST** method which is very similar but with proxy integration (**HTTP\_PROXY**) in this case I pass a request (you can find all the request examples in the postmanCollection folder) and API gateway "redirects" my request to **https://jsonplaceholder.typicode.com/posts** (note how in this integration types we have to specify the **httpMethod** not for reaching the API but instead how our API will reach another http backend endpoint) then we get the response. So, in summary this integration type can be very useful when working with existing backends reachable over http/https (is like chaining backends)

AWS integration //

Note: AWS\_PROXY integration is available just for lambda functions and other services but, if using S3, SNS or others, you must stick to AWS integration.

I created an endpoint in the template YAML **(/awsIntegration**) file which calls s3 API using the Action type set to PATH override, this will use the http method in the request that is forwarded from API gateway to the AWS service. (Note that the http verb of the client-API can be different from the one that happens in API-service)

There is a default service API endpoint that will get called from API gateway corresponding to the http verb, for example GET method will call the Get Object action by default (you can test this in postman)

This type of integration needs to use a ROLE with the necessary permissions so API gateway is allowed to call the specific AWS service actions, and the Role needs to have a trust relationship so it can be assumed by API gateway (a trust relationship is just a way to tell a role which entity can use it)

**Note that you can do this integration (with action set to path override) in two ways, send the request as is from the client or map it in the integration request as I did. The path overrides require the necessary parameters that are going to be forwarded to s3 to be in the path, but the client can send them as query string parameters since this will get mapped in the integration request from the query string parameters to the path parameters of the next request.**

Now what if you want to call an action that is not default to an http verb, well you can use the action type set to action override and specify any API action you want to call within a service then just structure the request as it needed in the request integration block or send it ready to go from the first client step. An example of this is the **/dynamoIntegration** resource I created which just inserts a record in a table, I used the **PutItem** action and passed it as it was from the client to the AWS service.

You can find examples of different AWS service integrations with s3 here:

[Tutorial: Create a REST API as an Amazon S3 proxy in API Gateway - Amazon API Gateway](https://docs.aws.amazon.com/apigateway/latest/developerguide/integrating-api-with-aws-services-s3.html)

<https://www.youtube.com/watch?v=dir1HtgbwzY>

As part of my research, I found this web to be useful while developing serverless solutions:

<https://serverlessland.com/patterns/apigateway-rest-s3-sam>

Here you can find all the actions available for any service (<service> api reference):

<https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_PutItem.html>

Maybe you noticed that to make this demo I had to create multiple resources like LogGroups, roles and even a DynamoDB table, these resources don’t have a specific syntax for SAM but since SAM is build on top of cloud Formation all the CFN syntax can be used here. You can find all about CFN resources here:

<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-template-resource-type-ref.html>

**In the YAML file specifying path or action looks something like this:**





**XLM requests:**

Enabling logging //

You can see how to enable logging step by step in the following link:

<https://docs.aws.amazon.com/apigateway/latest/developerguide/set-up-logging.html>

To enable logging you should create a AWS::Logs::LogGroupName resource and link it to the API resource (Destination: !GetAtt LogGroupApi.Arn) . A log Group is just a container of multiple log streams (which is just like a chinned event of logs) and Api Gateway will use the logGroup to push the events, like request and responses, I found this very useful to debug while doing this demonstration since some s3 actions needs xml request instead of the common application/json content type for request, since I didn’t have that much experience with application/xml this was very helpful.

**Note: Logging works by at the deployment stage level not at the full API level.**

**Note: While doing this demo I also found the AWS console very helpful while trying to figure out how some features works, so I had 3 stages one which is like a checkpoint (All that is there always works) another one which is to develop using CFN/SAM which could work partially and another one which is to manually make changes in the console (avoiding drifting at least in the main stages however a better way could have been having a separate api in which I don’t really care about drifting or a full manually created API) this manual changes were useful since I can export the API swagger definition in YAML format and then see what is the actual code representation of the configured settings within the console.**

**This is way faster since exploring and messing around with settings in the YAML file implicates a higher learning curve as well as slower deployment (SLOWER ONLY IF ERRORS ARE HAPPENING CONSTANTLY OTHERWISE CFN/SAM are way faster to deploy)**

**General considerations:**

Examples on how to call the endpoint/resource is within the postman collection in which you can see the URL, path/query parameters and any relevant information.

We will be using async/await syntax and the moment library just to get familiar with the use of external packages within the Serverless lambda architecture.

Note that you **can** send a body within a GET request in local testing, but it is not recommended and NOT possible in some production scenarios, use POST instead since some proxies, services or firewalls could not work with a body included withing a GET request, and this is the case when using API gateway.

You can find more about SAM here:

<https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-getting-started.html>

**Instructions:**

**Note: This first two steps are a brief overview of what I did to create the SAM template/project you don't need to do it and in the guide folder there is a file in which I explain in detail the features of SAM and CFN.**

**1-run:** sam init

(Remember that you need to have SAM CLI installed on your local environment)

<https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/install-sam-cli.html>

SAM needs some prerequisites to run, most of them are optional since they are to use some local testing features of SAM within docker containers on your local environment, but I recommend to follow the complete installation in the link above.

**2-Answer the prompts with the desired configuration, in my case it was:**

-AWS Quick Start Templates

-Hello world Example

-N

-13 (node 16.x)

-1 (zip)

-1 (Hello world Example)

-N

-y

-LambdaBasicsSAM

**The following steps are so you can deploy the SAM template and test it locally (local testing only available if you followed the SAM CLI prerequisites and installed docker).**

**1- To test the function by itself locally sit on the root directory of the SAM project and run:**

**sam local invoke <logicalFunctionName> -e .\events\<eventFunction>.json**

This will invoke the logical name of the function you have in the template.yaml file and will use the **-e** flag to specify a test event which you can find inside the events folder (remember that you need to allow docker to share the volumes of your pc, generally you get prompted automatically otherwise I will include a link in the references/important-links.txt file)

**2-We will test this demo with implicit and explicit API resource creation.**

<https://github.com/awslabs/serverless-application-model/blob/master/docs/internals/generated_resources.rst#api>

**To test the functions and the api locally run the following command.**

**sam local start-api --debug or sam local start-api**

IMPORTANT: Remember that SAM local testing/emulator is oriented towards lambda integration type so to test everything I did it would be better to just deploy the stack and test it over the internet.

After you run the command, it will give you an Ip with a port, use postman to test the path which is used to call our function:

Example (this path is specified in the template.yaml file within the definition of our resources):

**Running on http://127.0.0.1:3000/helloMulti/{name}**

**Note: Within the repo there is a folder called postman which will contain an exported JSON of the collections used for each demo, you can import it into your postman if you want, remember to create an environment for the collection and create the variable host within it with the value that SAM gave you in the previous step (http://127.0.0.1:3000/{resource})**

**3-sam package --template-file template.yaml --output-template-file sam.yaml --s3-bucket <bucket-name>**

**Note: the --s3-bucket flag should only be used if you want to specify a bucket in line otherwise it will pick the one in the samconfig.toml file, if neither the inline command nor the toml file define a bucket then a default bucket will be created automatically. (This bucket is used to store some necessary files for CFN to work)**

This command will transform your template.yaml file to another SAM yaml file which will have some variables resolved like the implicit resources or the URI of the function code that will no longer be on our local computer but an s3 bucket which you must create before running this command (I will explain all the details about SAM in the SAM guide)

**4-To deploy the stack run the following command:**

**sam deploy --template-file <the file created on step 3> --stack-name <YOUR STACK NAME>**

**5-Once you run the command**

Go and check CloudFormation console to see the status of the stack you are creating, alternatively you can see the outputs in the terminal. (Remember that CFN as well as the SAM frameworks offers useful tools for debugging)

**6-Once everything gets created** **successfully**

Go and check that a new Api was created and test it. I will not cover how to test it since the documents within the guide folder should give you enough understanding on how to do it.

**7-After testing**

Delete the stack from CFN and the bucket so you keep your environment in a clean status (This is optional since most of the demos will be built on top of each other and redeploying a Sam file will just update the resources from a stack if the stack name is the same)

**Additionally:** Keep in mind that this was a brief demo, and the purpose was not to go full detail into CFN or SAM and instead you should be focusing only on the behavior of the lambda and the Api Gateway services deployed through this demo.