**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)**

**This demo includes a better, simplified, structured demonstration of the same concepts found under the other-files/serverlessPOC2 folder.**

Explanation //

**Purpose:**

In this demo we are going to explore more API gateway features this time also using the JavaScript SDK version 3 to implement more complex logic as well as a more real scenario like writing to SQS.

ApiKeys/UsagePlan //

In this demo we have two APIs and one important thing to mention at this point is that if you have two API resources defined in the same template and you run **sam local start-api** the emulator will emulate only the first API resource it finds and there is no way to specify other API resource to emulate (There might be a change that that is actually possible but at the moment I couldn’t find how to do it and all indicators points that it is not possible, maybe in the SAM guide I will mention again if it is possible or not) It is also important to note that local testing will also affect resources in the cloud, for example a lambda function that creates an API key can be tested locally but it will create an API key in the AWS cloud, this “drawback” could be useful to test functions or use functions without being billed for the execution since all the execution is happening locally.

Moving on, the first API is an API that requires no authentication (**apiInitial**) this API has only two endpoints **/helloWorld** which is used to test the API easily and **/createAssignKey** which uses a lambda function as the backend to create an API key and link it to an existing usage plan.

**Note how I could have used AWS service integration type to create an API key but since the logic is more complex it was necessary to use a lambda function to implement the relationship between the API key and the usage plan. (I could have made two AWS service integration type endpoints one to create the api key and another one to link it to a usage plan but then I wouldn’t have been able to make validations or implement complex logic to create this resources)**

In the case of the **createAssignKey** lambda function I wanted to point that the implicit role creation for this lambda was not enough since I needed to grant the lambda permissions to perform actions on API gateway, therefore, I created an AWS::IAM::Role resource in the SAM template and assigned this role to the AWS::Serverless::Function resource.

**Note that in the CFN/SAM guides it will be really important to define the order in which resources are created, in this case for example SAM automatically defines that the role needs to be created before the function since the function depends on the role to exist. Also remember that this role needs to implement trust relationship so lambda function can assume the role.**

The function essentially just creates an API key and link it to a usage plan (that was created with SAM as a AWS::ApiGateway::UsagePlan resource which works for the **mainApi** API)

In the second API (**mainAPI**) theres only one endpoint which is /helloWorldMain, since this API and its API stage “testing” are linked to the usage plan we discussed previously this endpoint is configured to be accessible only if the x-api-key is provided in the header of the request otherwise it will return a forbidden 403 error.

Note how I used the same lambda function in different APIs, the **/helloWorldInitial** and **/helloWorldMain** endpoints both point to the same lambda function (remember to keep in mind the throttling limits for the lambda function if you decide to implement this in production environments)

**One feature that I found useful in the SAM template api resource is that you can define which specifics API keys are valid for an endpoint (In the security property), in the console the only way to achieve this is to would be at the UsagePlan level by setting manually a throttling limit to 0 for specific methods so this is a feature that I found available only if defining the API in the SAM/CFN template.**

**Texto

Descripción generada automáticamente con confianza baja**

SQS integration //

In this section I will just explain how to write to SQS using lambda as the backend, once again, I could do this with AWS service integration an save the lambda execution/intermediary but this limits the complexity of the logic I could implement (remember that using the request validators as well as mapping we could also implement some decent validation logic and if that happens to be enough there would be no necessity to use the lambda backend)

**Note: I will cover most of the features of SQS in future demos and the SQS guide but for now I’m sticking to just creating this resource in the SAM template file and then writing two endpoints to write and read from that queue.**

**A relevant subject I haven’t talked about yet is the structure of this yaml files, as you can see, they are getting bigger as we go through in the demos and therefore harder to maintain or understand, I will cover some techniques and good practices to handle this situation in the CFN/SAM guides.**

In this case I can’t implement the authorizer from within the AWS::Serverless:Function itself instead of the Api resource which is another way to achieve the same behavior that the /helloWorldMain endpoint has which needs an Api key to allow invocation, this is because I’m not using implicit Api resource creation but explicit and as I discussed earlier with the requiredParameters property in this case this properties should be specified by the API resource like I did in the above images.

**Important: remember that local testing with two APIs as far as I´m concerned is not supported by the SAM emulator, a way to achieve this behavior could be two structure the deployment with nested CFN/SAM stacks of two have two completely different SAM projects so you can run SAM local star-api for each project and use them in a separate port.**

For this part of the demo, I added an endpoint (/writeSQSmessage) to the mainApi Api which just runs a lambda function that sends a message to an SQS queue, note how I used environment variables for the lambda function to specify the ARN of the queue.

Texto

Descripción generada automáticamente con confianza media

What I needed to do here is essentially the same as the previous **createAssignKey** function which was to create a role that grants permission to write to SQS and instead of creating a usage plan resource I created an SQS queue resource within the SAM template once again reaffirming the concept that all CFN synthax is valid within SAM templates (this does not always works in the opposite way, in the SAM resource definition documentation you can find which properties have backwards compatibility with CFN)

To test all this demos you just need to run the requests in the GatewaySDK postman collection, remember to specify the API key in the x-api-key header or you will not be able to access the gateway resources.Aside from that you can confirm the api key creation as well as the SQS message being received by the queue through the AWS console.

**Note: I didn’t include the local requests in the postman collection because of the way I created the two APIs in the same template, however if you change the order in which the API are specified in the yaml file you can invoke and execute their functionality locally.**

**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.