



AWS SAM (Serverless Application Model) Overview

AWS SAM is an open-source framework which enables to create and deploy an application on AWS cloud. It simplifies the process by allowing developers to define serverless applications using AWS CloudFormation syntax, consolidating resources such as AWS Lambda functions, API Gateway endpoints, DynamoDB tables, and others into a single YAML or JSON file known as a `template.yaml` or `template.json`.

Let's explore some of the basic advantages of utilising AWS SAM for serverless application development and deployment.

I. Simplifies development and deployment of serverless applications

This simplifies the deployment process compared to manually configuring the resources through AWS Management Console

II. Seamless integration with AWS services Across Multiple Environments

It streamlines the development and deployment process, reducing the time and effort required to connect and configure AWS services manually. It also helps maintain consistency and reliability across deployments by ensuring that all necessary resources are provisioned and configured correctly.

III. Facilitates version control and adherence to Infrastructure as Code (IaC) principles

AWS SAM facilitates version control and adherence to Infrastructure as Code principles by providing a structured, version-controlled approach to defining, managing, and deploying serverless application infrastructure on AWS

AWS SAM Components

The AWS SAM (Serverless Application Model) comprises essential elements that simplify the creation and deployment of serverless applications on Amazon Web Services (AWS). These components encompass:

- I. **Template:** SAM Template is the core component of AWS SAM Infrastructure as it acts as the blueprint as it defines resources and configurations. Template script can be in either json or yaml format (template.json/template.yaml).
- II. **SAM CLI:** It enables developers to execute a set of commands to build, test and deploy your application. It basically used AWS CLI underneath to deploy your application on a respective account.
- III. **SAM Local:** SAM local is feature that enables to test your code without being deployed on aws cloud
- IV. **SAM Build:** SAM build provides you with the feature to build your app with all the required dependencies defined in requirements.txt file.
- V. **SAM Templates:** AWS SAM comes up with basic blueprint to start with your application providing basic folder structures and files
- VI. **SAMCONFIG.TOML :** A configuration file used by AWS cli to store settings and parameters for deploying serverless applications. This file is generated at the time of execution where you can define values against the keys which are defined in the parameters sections of the template file along with the default keys such as stack-name, aws region.

This is a basic overview of AWS SAM, for further details you can visit official AWS SAM [documentation](#)

AWS SAM Installation Guide:

To start with AWS SAM you need to have following requirements:

- An AWS account is required with access key ID and secret access key of the account for a user with cli access. [link](#)
- AWS SAM CLI installation
You can download cli from official aws [website](#) as per your operating system and validate it by executing **sam --version** command in your cmd
- Following is the [link](#) to an example aws sam application by which you can testify your cli configuration and access. Above link will lead you to aws official documentation where a complete flow is defined to execute the example script with respective commands.

AWS SAM Customization:

In this customization we will look into things in which we will making changes in template.yaml, samconfig.toml, adding python scripts and dependencies.

Multiple Development Environment

- **Why do we need multiple development environments?**

Creating multiple environments is a key feature of aws sam as it allows to replicate same configurations and components across multiple regions without errors. It basically complements the working process of developing the app in dev environment and on successful development it moved to production environment without extra effort just a minor change in command.

- **How to integrate multiple development environments?**

To integrate an environments we need to pass on configurations of different environments such as below:

```
[env.deploy.parameters]
stack_name = "sam-dev-app-run"
resolve_s3 = true
s3_prefix = "sam-dev-app-run"
region = "us-east-1"
confirm_changeset = true
capabilities = "CAPABILITY_IAM"
parameter_overrides = "securityGroup=\"sg-XXX\"
                        subnet1=\"subnet-XXX\"
                        subnet2=\"subnet-XXX\"
                        listingDatabase=\"ylopo/listing/db/prod\"
                        rdsDatabase=\"ylopo/rds/db/prod\"
image_repositories = []
```

Above is the format how the configuration is defined in samconfig.toml file.

In the above payload other than parameter_overrides all variables are default. So parameter_overrides are parameters which we defined in our template.yaml script for the values which are being used in the configurations of our applications and differs on the basis of environment.

- **How we define them in our template.yaml script**

So in our template file we define a section name parameter and in that section we define variable and its type such as follows:

Parameters:

SecurityGroup:

Type: String

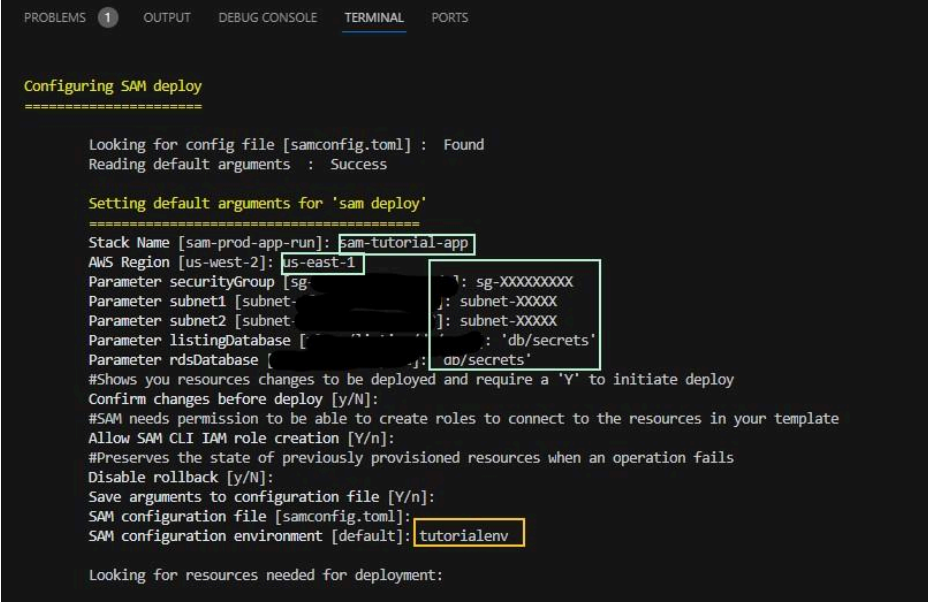
For further details visit the following [link](#)

- **How to create multiple environments and define values against them**

In your cmd execute

- `sam deploy --guided`

Add values against the variable as shown in the following screenshot



```
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS

Configuring SAM deploy
=====

Looking for config file [samconfig.toml] : Found
Reading default arguments : Success

Setting default arguments for 'sam deploy'
=====
Stack Name [sam-prod-app-run]: sam-tutorial-app
AWS Region [us-west-2]: us-east-1
Parameter securityGroup [sg-XXXXXXX]: sg-XXXXXXXXXX
Parameter subnet1 [subnet-XXXXXX]: subnet-XXXXXX
Parameter subnet2 [subnet-XXXXXX]: subnet-XXXXXX
Parameter listingDatabase [db/secrets]: db/secrets
Parameter rdsDatabase [db/secrets]: db/secrets
#Shows you resources changes to be deployed and require a 'Y' to initiate deploy
Confirm changes before deploy [y/N]:
#SAM needs permission to be able to create roles to connect to the resources in your template
Allow SAM CLI IAM role creation [Y/n]:
#Preserves the state of previously provisioned resources when an operation fails
Disable rollback [y/N]:
Save arguments to configuration file [Y/n]:
SAM configuration file [samconfig.toml]:
SAM configuration environment [default]: tutorialenv

Looking for resources needed for deployment:
```

Here in the bottom highlighted in the yellow box is the name of the environment

- **How define environment when deploying application**

Following is the command to specify environment

- `sam deploy --config-env tutorialenv`

CREATION OF RESOURCES IN AWS SAM

Before we start with aws sam resources kindly go through this [link](#) as it elaborates other key components which we will not be discussing below but are helpful for context and better understanding.

A resource is an infrastructure component/aws service which is used to define it in a template file.

Each resource has a Type which basically defines the service and its component such as for lambda type will be as follow: **Type: AWS:Serverless:Function**

Similarly there is another tag name Properties. Which contains the configurational information for that particular component. Below is an example of lambda function configuration which we will be discussing one by one.

1. CodeUri: This tag defines where your app.py file is placed, which basically contains lambda code. `lambdas/mls/locking_lambda/`
2. Handler: This tag defines what is your main function name.(this basically refers to lambda_handler function as it drives the code). Name could be different `lambda_handler.handler`
3. FunctionName: In this tag you basically defines the name of lambda function which you want to show on AWS Console `mls-locking-func`
4. Runtime: Lambda language and its versions `python3.9`
5. Timeout: Time out limit of lambda
6. MemorySize: Memory size of lambda
7. Role: IAM role you want to assign to the lambda function `!GetAtt MlsLambdaRole.Arn`
8. VpcConfig: In this section you need to provide information regarding your security group and subnets when you have to create private lambda
9. Layers: In this section you attach layers to your function
10. Environment Variable: Here you define your environment variables

LockingLambdaFunction:

Type: `AWS::Serverless::Function`

Properties:

CodeUri: `lambdas/mls/locking_lambda/`

Handler: `lambda_handler.handler`

FunctionName: `mls-locking-func`

Runtime: `python3.9`

Timeout: `300`

MemorySize: `128`

Role: `!GetAtt MlsLambdaRole.Arn`

VpcConfig:

SecurityGroupIds:

- `!Ref securityGroup`

SubnetIds:

- `!Ref subnet1`

- `!Ref subnet2`

Layers:

- `!Ref awsWrangle`

- `!Ref helperLayer`

- `!Ref psycopg2`

Environment:

Variables:

`sourcelds: "796,359,483,759"`

`listingDatabase: !Ref listingDatabase`

`rdsDatabase: !Ref rdsDatabase`

Before we move we need to discuss few keywords which were defined earlier

- **!GetAtt** a function which used to fetch ARN of the resources created in your template such IAM role. Key point to note here is that !GetAtt will be followed by the name of the resource in the template such as if we want fetch arn of above lambda then this will look like below not the name specified in properties

`!GetAtt LockingLambdaFunction.Arn`

- **!Ref** a function used to reference other resource created in your template
- **!Sub** this a substitution function, by this you get values set against the variable
- - this implies that a list of values are passed against a variable

For Further details following are the links to AWS documentations for services used in our IDX serverless application.

- [Lambda](#)
- [Stepfunction](#)
- [IAM](#)
- [Layers](#)
- [LogGroup](#)

Please note that this document serves as a foundational introduction to AWS SAM, aimed at acquainting you with its fundamental concepts. For the most recent and detailed information, it is advisable to consult the official AWS SAM documentation available on their website.

To enhance comprehension and mastery of AWS SAM, it is highly recommended to diligently follow the AWS SAM documentation, allowing for a thorough understanding of its concepts and features.