AWS CloudFormation & Terraform

When it comes to automating deployments on the Cloud, both AWS CloudFormation and Terraform are excellent tools that each come with their own set of advantages.

CloudFormation, being AWS's proprietary tool, is a natural go-to for AWS users looking to rapidly deploy and automate their infrastructure on the Cloud. AWS's preconfigured CloudFormation stacks provide users with the ability to quickly and easily deploy their AWS infrastructure with the click of a button.

However Terraform - an open-source infrastructure as code software tool created by HashiCorp - is an equally desirable tool for companies requiring infrastructure automation. Terraform's open-source nature along with its wide user pool and complete ecosystem of products make it a very appealing tool for companies that prioritize a certain measure of flexibility and control over their cloud deployments.

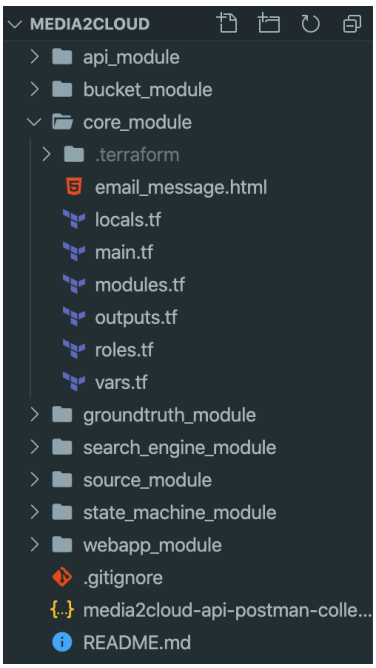
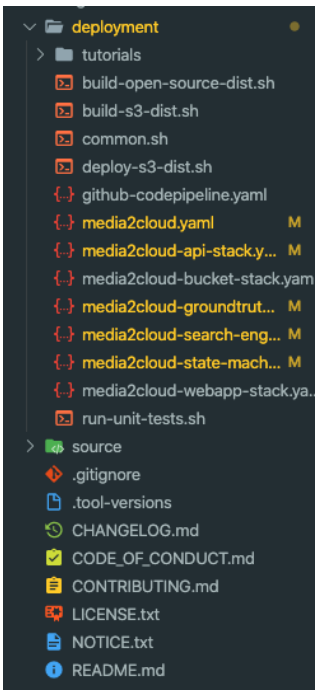
"Many of TrackIt's clients have standardized on Terraform for its broad coverage of services and it's broad open-source user community. TrackIt has extensive experience with both it and CloudFormation and often will make a recommendation of one or the other." - Brad Winett, President, TrackIt

One of TrackIt's customers had standardized on Terraform as its provisioning tool and wanted TrackIt's help in implementing AWS Media2Cloud using Terraform. There was no pre-existing Terraform module that the TrackIt team could leverage to quickly implement Media2Cloud, so TrackIt's team translated the CloudFormation stack into Terraform modules itself.

This whitepaper describes our team's approach to translating Media2Cloud CloudFormation stacks into Terraform modules.

Preliminary Step: Cleaning Up the Folder & Splitting the CloudFormation Stacks

The first step of the process was to clean up the deployment folder in the Media2Cloud repository. We moved all the yaml files related to the CloudFormation stack into a separate folder in order to follow Terraform best practices and to make it easier to maintain.

	
Terraform	Cloudformation

The next step of the process was to split each of the CloudFormation stacks into separate files which would then be converted to Terraform modules.

Refactoring CloudFormation Stacks

One of the primary challenges during the translation was to find an equivalent for AWS CloudFormation Custom Resources within Terraform. To address this issue, it was necessary for us to properly understand what Custom Resources do; to summarize, they are used to call Lambda functions at specific provisioning times.

A Custom Resource can be configured in the following manner in Terraform:

```
{
   "RequestType" : "Create",
   "ResponseURL" : "http://pre-signed-S3-url-for-response",
   "StackId" :
"arn:aws:cloudformation:us-west-2:123456789012:stack/stack-name/guid",
   "RequestId" : "unique id for this create request",
   "ResourceType" : "Custom::TestResource",
```

```
   "LogicalResourceId" : "MyTestResource",
   "ResourceProperties" : {
      "Name" : "Value",
      "List" : [ "1", "2", "3" ]
   }
}
```

This JSON object triggers a specific Lambda function deployed by CloudFormation. The most important part of the JSON object is the RequestType which informs the Lambda function what “state” Terraform is in. (To learn more about states, please refer to the section below titled “States”). For example, when the state is "Create", the Lambda function will know that the infrastructure is being deployed. Another important property is ResourceProperties, which is used to pass custom data to the Lambda function.

Terraform has a specific data source called "aws_lambda_invocation" which is used to trigger a Lambda function. As you can see in the following snippets, we deployed the Lambda function and used this data source to call the Lambda function at the creation of the infrastructure.

```
resource "aws_lambda_function" "custom_resource_function" {
 depends_on =
[aws_iam_role_policy_attachment.attach_custom_resource_execution_role_policy]
 function_name = "custom_resource_function"
 description   = "(${local.solution.project.id}) custom resources"
 runtime       = "nodejs10.x"
 memory_size   = 128
 timeout       = 900
 handler       = "index.Run"
 role          = aws_iam_role.custom_resource_execution_role.arn

 filename = module.source.custom_resources
}
```

Deploying the Lambda function

```
data "aws_lambda_invocation" "iot_url" {
 depends_on    = [aws_lambda_function.custom_resource_function]
 function_name = aws_lambda_function.custom_resource_function.function_name

 input = <<JSON
 {
```

```
  "RequestType": "Create", // HERE
  "ResourceProperties": {
    "FunctionName": "IotEndpoint",
    "ServiceToken": "${aws_lambda_function.custom_resource_function.arn}"
  }
}
JSON
}
```

Calling the Lambda function

```
resource "null_resource" "detach_iot" {
 depends_on = [aws_iot_policy.iot_thing_policy,
aws_lambda_function.custom_resource_function]
 triggers = {
   region        = var.region
   function_name = aws_lambda_function.custom_resource_function.function_name
   service_token = aws_lambda_function.custom_resource_function.arn
   iot_policy    = aws_iot_policy.iot_thing_policy.name
 }
 provisioner "local-exec" {
   when    = destroy
   command = <<COMMAND
     aws lambda invoke \
     --cli-binary-format raw-in-base64-out \
     --log-type Tail \
     --region ${self.triggers.region} \
     --function-name ${self.triggers.function_name} \
     --payload '{
       "RequestType": "Delete",
       "ResourceProperties": {
         "FunctionName": "IotDetachPolices",
         "ServiceToken": "${self.triggers.service_token}",
         "IotThingPolicy": "${self.triggers.iot_policy}"
       }
     }' \
     delete-iot.json
   COMMAND
 }
}
```

Calling the Lambda function at destroy

By default, Terraform runs the `data "aws_lambda_invocation"` data source all the time, but we wanted it to run it *only* when the “state” is ‘destroy’. As a workaround, we used the provisioner "local-exec" which has the following condition: `when = destroy` hence allowing the lambda function to be triggered solely when the “state” is ‘destroy’.

“States”

In CloudFormation, stacks have specific status codes (“states”) that describe their life cycles. The following are examples of states that CloudFormation returns:

`CREATE_COMPLETE`	Successful creation of one or more stacks.
`CREATE_IN_PROGRESS`	Ongoing creation of one or more stacks.

In Terraform we don’t have access to these “states” because Terraform directly uses the AWS API and has its own life cycle mechanism. This presents a challenge since various lambda’s source code use these “states” to perform some processes. For instance, there is a lambda function that is triggered in CloudFormation only during the ‘destroy’ state to detach a specific policy from a resource. This proved to be a slight challenge while we were implementing the same Lambda function in Terraform. In Terraform we execute the command “terraform apply” for the first deployment and for any updates done after that. We realized that the prior lambda function was being triggered each time the command “terraform apply” was executed, and we wanted this function to be triggered only when the command “terraform destroy” is executed.

In order to address this issue, we added the “RequestType” attribute to the payload that is sent to our lambda function.

Here is a good example of the usage of “null_resource” resource to trigger the correct lambda function related to the “custom_resource” based on the deployment “state”:

```
resource "null_resource" "send_config" {
 depends_on = [null_resource.create_index]
 triggers = {
   region          = var.region
   function_name   = var.function_name
   service_token   = var.custom_resource_arn
   solution_id     = var.solution_id
   solution_uuid   = var.solution_uuid
   version         = var.solution_version
   anonymous_usage = var.anonymous_usage
   cluster_size    = var.cluster_size
 }
```

```
provisioner "local-exec" {
  command = <<COMMAND
    aws lambda invoke \
    --cli-binary-format raw-in-base64-out \
    --log-type Tail \
    --region ${self.triggers.region} \
    --function-name ${self.triggers.function_name} \
    --payload '{
      "RequestType": "Create",
      "ResourceProperties": {
        "FunctionName":   "SendConfig",
        "ServiceToken":   "${self.triggers.service_token}",
        "SolutionId":     "${self.triggers.solution_id}",
        "SolutionUuid":   "${self.triggers.solution_uuid}",
        "Version":        "${self.triggers.version}",
        "AnonymousUsage": "${self.triggers.anonymous_usage}",
        "ClusterSize":    "${self.triggers.cluster_size}"
      }
    }' \
    send_config.json
  COMMAND
}

provisioner "local-exec" {
  when    = destroy
  command = <<COMMAND
    aws lambda invoke \
    --cli-binary-format raw-in-base64-out \
    --log-type Tail \
    --region ${self.triggers.region} \
    --function-name ${self.triggers.function_name} \
    --payload '{
      "RequestType": "Delete",
      "ResourceProperties": {
        "FunctionName":   "SendConfig",
        "ServiceToken":   "${self.triggers.service_token}",
        "SolutionId":     "${self.triggers.solution_id}",
        "SolutionUuid":   "${self.triggers.solution_uuid}",
        "Version":        "${self.triggers.version}",
```

```
        "AnonymousUsage": "${self.triggers.anonymous_usage}",
        "ClusterSize":    "${self.triggers.cluster_size}"
      }
    }' \
    delete_send_config.json
  COMMAND
 }
}
```

Refactoring the Source Code

We wanted to build source codes for the lambda functions. For this, we created a specific module in Terraform designated for source code building and archiving. We took all the sources and moved them inside this module. We refactored a bash script that was used to install dependencies and interpolate strings inside CloudFormation stacks. We removed the following variables and set them directly in Terraform:

```
#
# zip packages
#
## Custom resource package
PKG_CUSTOM_RESOURCES=

## Lambda layer package(s)
LAYER_AWSSDK=
LAYER_MEDIAINFO=
LAYER_CORE_LIB=
LAYER_IMAGE_PROCESS=
LAYER_FIXITY_LIB=
# note: core-lib for custom resource
CORE_LIB_LOCAL_PKG=
## modular workflow package(s)
PKG_S3EVENT=
PKG_INGEST=
PKG_ANALYSIS_MONITOR=
PKG_AUDIO_ANALYSIS=
PKG_VIDEO_ANALYSIS=
PKG_IMAGE_ANALYSIS=
```

```
PKG_DOCUMENT_ANALYSIS=
PKG_GT_LABELING=
PKG_API=
PKG_ERROR_HANDLER=
PKG_WEBAPP=
```

The following were the only variables that we kept for source building:

```
NODEJS_VERSION=$(node --version)
DEPLOY_DIR="$PWD"
SOURCE_DIR="$DEPLOY_DIR"
TEMPLATE_DIST_DIR="$DEPLOY_DIR/global-s3-assets"
BUILD_DIST_DIR="$DEPLOY_DIR/regional-s3-assets"
TMP_DIR=$(mktemp -d)
```

We used these variables in the following way:

```
#
# @function build_image_process_layer
# @description
#   build layer packages and copy to deployment/dist folder
#
function build_image_process_layer() {
 echo "------------------------------------------------------------------------------"
 echo "Building image-process layer package"
 echo "------------------------------------------------------------------------------"
 pushd "$SOURCE_DIR/layers/image-process-lib" || exit
 LAYER_IMAGE_PROCESS=$(grep_zip_name "./package.json")
 npm install
 npm run build
 npm run zip -- "$LAYER_IMAGE_PROCESS" .
 cp -v "./dist/${LAYER_IMAGE_PROCESS}" "$BUILD_DIST_DIR"
 popd
}
```

After this refactoring, we created a “null_resource” resource to build the sources and archive them:

```
resource "null_resource" "build_archives" {
 provisioner "local-exec" {
   command = "pushd ${path.module} && ./build-s3-dist.sh && popd"
 }
```

```
}
```

Once the archives were built, we output the path:

```
output "aws_sdk_layer" {
 value       = "${path.module}/regional-s3-assets/aws-sdk-layer.zip"
 description = "AWS SDK Layer zip file"
 depends_on  = [null_resource.build_archives]
}
```

And we use the outputs from the archiving modules in the other modules that need them:

```
module "bucket" {
 source = "../bucket_module"

 region        = var.region
 account_id    = data.aws_caller_identity.current.account_id
 force_destroy = var.s3_force_destroy

 solution_id     = local.solution.project.id
 root_stack_name = var.root_stack_name
 solution_uuid   = random_uuid.solution_uuid.result
 randomized_name = local.randomized_name
 anonymous_usage = var.anonymous_usage

 bucket_prefix  = local.randomized_name
 sns_topic_arn  = aws_sns_topic.sns_topic.arn
 iot_host       = jsondecode(data.aws_lambda_invocation.iot_url.result)["Endpoint"]
 iot_topic      = "${local.solution.project.id}-${var.root_stack_name}/status"
 aws_sdk_layer  = aws_lambda_layer_version.aws_sdk_layer.arn
 core_lib_layer = aws_lambda_layer_version.core_lib_layer.arn

 s3_event_archive = module.source.s3_event
}
```

```
resource "aws_lambda_function" "on_object_created" {
 function_name = "${var.solution_id}-${var.root_stack_name}-s3event"
 description   = "(${var.solution_id}) OnObjectCreated starts an ingest workflow"
 runtime       = "nodejs10.x"
 memory_size   = 128
 timeout       = 900
```

```
  handler       = local.function.handler.on_object_created
  role          = aws_iam_role.execution_role.arn

  filename = var.s3_event_archive

  layers = [
    var.aws_sdk_layer,
    var.core_lib_layer
  ]

  environment {
    variables = {
      ENV_SOLUTION_ID     = var.solution_id
      ENV_STACKNAME       = var.root_stack_name
      ENV_SOLUTION_UUID   = var.solution_uuid
      ENV_ANONYMOUS_USAGE = var.anonymous_usage
      ENV_SNS_TOPIC_ARN   = var.sns_topic_arn
      ENV_IOT_HOST        = var.iot_host
      ENV_IOT_TOPIC       = var.iot_topic
    }
  }
}
```

Blockers & Workarounds

During the translation phase, we experienced some difficulties with the CloudFormation stack because of its use of custom resources. In order to address these issues, we had to develop a number of workarounds and refactor some lambda functions.

For example, the custom_resource_function handles:

- The IoT detach policy function (used for notifications on the web application)
- The configure workteam function (used for the labeling jobs)
- The create/delete custom vocabulary function (used by Amazon Transcribe)
- The create index function (used to generate the manifest.js file used by the web application)

We used Terraform's "null_resources" to execute script files to build and archive sources, provision lambda functions, and for the web application (see an example below).

```
resource "null_resource" "build_archives" {
 provisioner "local-exec" {
   command = "pushd ${path.module} && ./build-s3-dist.sh && popd"
 }
}
```

Redundant lambda functions that were previously used to format strings were also removed because these lambda functions could be handled by Terraform directly. For example:

- The sanitize functions (that were replaced by variables and regular expressions check)
- The randomize name function (that were replaced by random_uuid Terraform resource)

The “null_resources” resource has also been used to execute certain functions only during the “apply” state or only during the “destroy” state of Terraform.

For all the resources that were previously created by the ‘Custom Resources’ resource within CloudFormation, we created equivalents within Terraform. For example, for the user pool domain of AWS Cognito (which is normally created within the AWS API while using CloudFormation) we used the AWS Cognito Domain resource on Terraform.

We also discovered an internal bug in the AWS platform that leaves phantom resources. This makes CloudFormation stacks or Terraform modules impossible to redeploy inside the region where the phantom resource has been created. AWS is currently working on a fix.