CLOUD MAGAZINE | N°ONE

binx.io

proudly part of Xebia Group

Some of our clients

Moving to the cloud is the single most significant technology shift with business impact a company will face over the next decade. At binx.io, we are passionate about making IT systems faster, better and cheaper using Cloud Technology.

There simply is no way back. Soon, all technology will be served from Cloud Platforms. Cloud services like image recognition and voice API's will make your consumer apps shine. The speed in which you serve new functionality will make your customers smile. Make no mistake, cloud is not a choice, it is a necessity for those who want to win.

In this very first edition of Cloud Magazine, we share a few articles about different cloud tools, including AWS CloudFormation and Lambdas, as well as a few customer stories. This Cloud Magazine is just one of the ways for us to share our experiences and insights, so that organizations develop the ability to adapt faster.

At binx.io you get to work together with the best, inspiring cloud technologists. If you are a technologist with an insatiable appetite for new skills, who works to make a difference and to create delighted customers, then make sure to check out our website; binx.io

We hope that you enjoy reading this magazine and would love to hear your feedback.

Mark van Holsteijn
CTO at binx.io

Deploying MySQL users and databases with AWS CloudFormation

Although CloudFormation is very good in creating MySQL database servers with Amazon RDS, it only contains a single user which has full control over the entire server. The mundane task of creating normal users and database schemas is not supported. With this Custom Resource Provider you can deploy a MySQL user with its own database schema as part of the CloudFormation template.

This plugin creates a user and a database schema and grants the user full access to the newly created schema, with grant option. This allows the application to create all the resources it requires and grant privileges to other users. The password for the user can be specified directly or taken from the AWS Parameter Store.

How does it work?

It is quite easy: you specify a CloudFormation resource of the Custom::MySQLUser, as follows:

```
KongUser:
  Type: Custom::MySQLUser
  DependsOn: KongPassword
  Properties:
    Name: kong
    PasswordParameterName: /mysql/kong/password
    WithDatabase: true
    DeletionPolicy: Retain
    Database: # the server to create the new user or
    database in
      Host: mysql
      Port: 3306
      Database: root
      User: root
      PasswordParameterName: /mysql/root/password
    ServiceToken: !Sub 'arn:aws:lambda:${AWS::Region}:
    ${AWS::AccountId}:function:binxioio-cfn-mysql-user-
    provider-vpc-${AppVPC}'

KongPassword:
  Type: Custom::Secret
  Properties:
    Name: /mysql/kong/password
    ServiceToken: !Sub 'arn:aws:lambda:${AWS::Region}:$
    {AWS::AccountId}:function:binxio-cfn-secret-provider'
```

After the deployment, the MySQL user 'kong' has been created together with a matching database 'kong'. The password for the user 'kong' was generated by the secret provider and stored as the parameter **'/mysql/kong/password'**. The password for the root database user has been obtained by querying the Parameter / mysql/root/password. If you just want to create a user with which you can login to the MySQL database server, without a database, specify **WithDatabase** as **false**.

The password for the user and the database connection can be specified directly (**Password**) or taken from the AWS Parameter Store (**PasswordParameterName**). We recommend to always use the Parameter Store.

By default WithDatabase is set to **true**, which means that a database schema is created with the same name as the user. If you only wish to create a user, specify **false**. When the resource is deleted, by default the user account is locked (RetainPolicy set to **Retain**). If you wish to delete the user (and the data), set RetainPolicy to **drop**.

If a user with the same name already exists, the user is "adopted" and it's password is changed. If **WithDatabase** is specified and a database/schema with the same name already exists, the user is granted all permissions on the database.

Installation

To install this Custom Resource, type:

```
export VPC_ID=$(aws ec2  --output text --query
'Vpcs[?IsDefault].VpcId' describe-vpcs)
export SUBNET_ID=$(aws ec2 --output text --query
Subnets[0].SubnetId \
    describe-subnets --filters Name=vpc-id,Values=$VPC_ID)
export SG_ID=$(aws ec2 --output text --query "Security-
Groups[*].GroupId" \
    describe-security-groups --group-names default  --filters
    Name=vpc-id,Values=$VPC_ID)

aws cloudformation create-stack \
    --capabilities CAPABILITY_IAM \
    --stack-name cfn-mysql-user-provider \
    --template-body file://cloudformation/cfn-custom-resource-
    provider.json  \
    --parameters \
    ParameterKey=VPC,ParameterValue=$VPC_ID \
    ParameterKey=Subnet,ParameterValue=$SUBNET_ID \
    ParameterKey=SecurityGroup,ParameterValue=$SG_ID

aws cloudformation wait stack-create-complete
--stack-name cfn-mysql-user-provider
```

Note that this uses the default VPC, subnet and security group. As the Lambda function needs to connect to the database, you will need to install this custom resource provider for each vpc that you want to be able to create database users.

If you have any questions, do not hesitate to contact me.

Mark van Holsteijn
CTO
markvanholsteijn@binx.io

This CloudFormation template will use our pre-packaged provider from
s3://binxio-public/lambdas/cfn-mysql-user-provider-latest.zip

If you have not done so, please install the secret provider too.

```
cd ..
git clone https https://github.com/binxio/cfn-secret-provider.git
cd cfn-secret-provider
aws cloudformation create-stack \
    --capabilities CAPABILITY_IAM \
    --stack-name cfn-secret-provider \
    --template-body file://cloudformation/cfn-custom-re-
    source-provider.json
aws cloudformation wait stack-create-complete  --stack-name
cfn-secret-provider
```

Demo

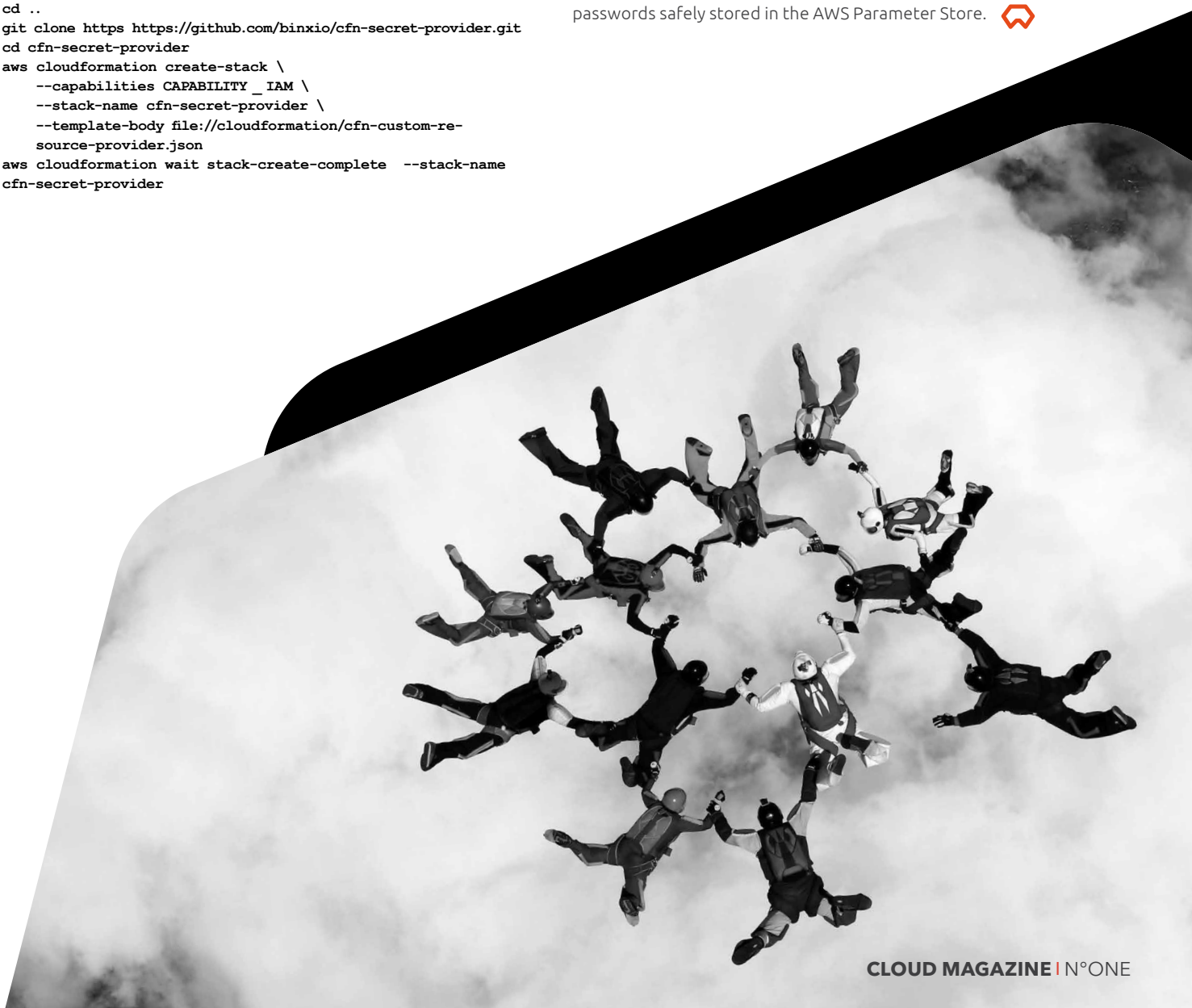
To install the simple sample of the Custom Resource, type:

```
aws cloudformation create-stack --stack-name cfn-mysql-
user-provider-demo \
    --template-body file://cloudformation/demo-stack.json
aws cloudformation wait stack-create-complete  --stack-name
cfn-mysql-user-provider-demo
```

It will create a MySQL database for demoing purposes, so it is quite time consuming…

Conclusion

With this solution MySQL users and database schemas can be provisioned just like the RDS instance, while keeping the passwords safely stored in the AWS Parameter Store.



A Serverless Version of Cfn-flip

The original version of cfn-flip is a command line tool that converts CloudFormation templates from JSON to YAML and vice versa. This tool has existed for some time and does the job magnificently. Most people find YAML more readable. I belong to the minority that finds JSON easier to read.

In my experience new idea's aren't born trying to think up what doesn't yet exist, but rather by observing something *lacking* and building something around that. This was definitely the case for *serverless cfn-flip*. The idea to create a serverless version of the tool that could do more than just convert the two formats was born when I noticed cfnflip.com was not registered.

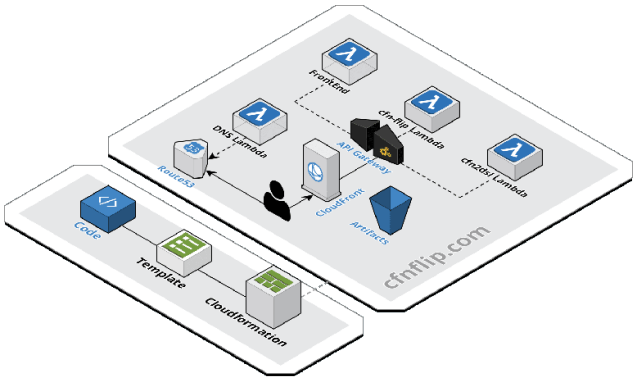
Objective

Before I wrote a single line of code I thought of what I wanted to achieve and narrowed it down to the following:

1. As little as possible interaction with the AWS console!
2. Created fully in CloudFormation.
3. As I am a Ruby fanatic, it must output to CfnDsl as well.
4. Each conversion function in its own Lambda function.
5. With pretty colours (syntax highlighting)
6. Backed with API Gateway, of course.
7. Giving back to the community by open sourcing the entire project.

Architecture

CfnFlip consists of an S3 bucket, four Lambda functions, an API Gateway, a CloudFront distribution and Route53. I've crafted the architecture to provide you with a visual overview:



CloudFormation Dependency Stack

I really like how the serverless framework creates an S3 bucket on the fly. It can store your artifacts there without the need to perform any manual bucket creation in the console or with the CLI. Inspired by that simple yet eloquent awesomeness I decided that this project needed that kind of magic as well. This came in the form of a dependency stack where I create some immutable resources that subsequent stacks import. The dependency stack took this form:

```
AWSTemplateFormatVersion: '2010-09-09'
Parameters:
  DomainName:
    Type: String
    Description: Domain name for the hosted zone
    Default: cfnflip.com
  ProjectName:
    Type: String
    Description: Name of your project, e.g. cfnflip
    Default: CfnFlip
```

I've used parameters here so you can easily re-use this snippet for another project. It takes the domain name and project name as parameters. Then onto the resources:

```
CloudformationBucket:
    Type: AWS::S3::Bucket
```

We let CloudFormation create a bucket with a dynamic name. The name will be outputted and exported. Then we let CloudFormation create our HostedZone.

```
HostedZone:
  Type: AWS::Route53::HostedZone
  Properties:
    HostedZoneConfig:
      Comment: !Join
        - ''
        - - 'HostedZone for '
          - !Ref 'DomainName'
    Name: !Ref 'DomainName'
```

Next up is a Lambda function. I wanted to automatically configure the Route53 domain name (cfnflip.com) with the DNS servers of the dynamically created Route53 HostedZone, so I would have to have the least amount of interaction with the AWS console. A custom resource invokes this Lambda function and passes on the DNS servers and then proceeds to configure the domain name using the NodeJS AWS SDK:

```
LambdaFunction:
  Type: AWS::Lambda::Function
  Properties:
    Handler: index.handler
    Role: !GetAtt 'LambdaFunctionRole.Arn'
    Code:
      ZipFile: !Join
        - "\n"
        - - var AWS = require('aws-sdk');
          - 'AWS.config.update({region: ''us-east-1''});'
          - var response = require('cfn-response');
          - exports.handler = function(event, context) {
          - '    var nameservers = event.ResourceProperties.
                 Nameservers;'
          - '    var domainname  = event.ResourceProperties.
                 HostedZone;'
          - ''
```

```
        - '    console.log(nameservers);'
        - '    console.log(domainname);'
        - '    if (event.RequestType == ''Delete'') {'
        - '      var responseData = { Value: ''Go ahead.'' };'
        - '      response.send(event, context, response.
                 SUCCESS, responseData);'
        - '  }'
        - '  else'
        - '  {'
        - ''
        - '      var route53domains = new AWS.Route53-
                 Domains();'
        - '      var ServerList = [];'
        - '      nameservers.forEach(function(server){'
        - '        ServerList.push({'
        - '          Name: server'
        - '        })'
        - '      });'
        - '      route53domains.updateDomainNameservers('
        - '          {'
        - '              DomainName: domainname,'
        - '              Nameservers: ServerList'
        - '          }, function(err, data) {'
        - '              if (err) console.log(err,
                         err.stack);'
        - '              else     console.log(data);'
        - '              var responseData = { Value: '
                         'Hurray!'' };'
        - '              response.send(event, context,
                         response.SUCCESS, responseData);'
        - '          }'
        - '      )'
        - '  }'
        - '};'
    Runtime: nodejs6.10
    MemorySize: '128'
    Timeout: '25'
  DependsOn: LambdaFunctionRole
```

Of course our Lambda function needs permissions. Our execution role permits the Lambda function to update the DNS servers and to log to CloudWatch:

```
LambdaFunctionRole:
  Type: AWS::IAM::Role
  Properties:
    Path: /
    AssumeRolePolicyDocument:
      Version: '2012-10-17'
      Statement:
        - Effect: Allow
          Principal:
            Service: lambda.amazonaws.com
          Action:
            - sts:AssumeRole
    Policies:
      - PolicyDocument:
          Version: '2012-10-17'
          Statement:
            - Action:
                - logs:CreateLogGroup
                - logs:CreateLogStream
                - logs:PutLogEvents
              Effect: Allow
              Resource: arn:aws:logs:*:*:*
            - Action:
                - route53domains:UpdateDomainNameservers
              Effect: Allow
              Resource: '*'
        PolicyName: Route53LambdaUpdateRole
```

Finally, we define a custom resource and pass on the DNS servers to our Lambda function:

```
UpdateRoute53:
  Type: Custom::UpdateRoute53
  Properties:
    HostedZone: !Ref 'DomainName'
    Nameservers: !GetAtt 'HostedZone.NameServers'
    ServiceToken: !GetAtt 'LambdaFunction.Arn'
  DependsOn:
    - LambdaFunction
    - HostedZone
```

We now have a HostedZone and an S3 bucket that should never change. We output and export these immutable values for our application stack to use:

```
Outputs:
  HostedZoneId:
    Value: !Ref 'HostedZone'
    Export:
      Name: !Join
        - ''
        - - !Ref 'ProjectName'
          - HostedZoneId
  HostedZoneName:
    Value: !Ref 'DomainName'
    Export:
      Name: !Join
        - ''
        - - !Ref 'ProjectName'
          - HostedZoneName
  CloudformationBucket:
    Value: !Ref 'CloudformationBucket'
    Export:
      Name: !Join
        - ''
        - - !Ref 'ProjectName'
          - CloudformationBucket
```

You can import an exported value in another stack by referring to it like this: !ImportValue 'MyExportedValueName'

A word of caution: If you use an exported value in another stack you can't modify the value of what you've exported until no stack refers to it anymore. In other words, use exports only for data that never ever changes.

The Lambda functions

cfn-flip is a Python script that ordinarily takes a flat file as argument and transforms it to either YAML or JSON, whereas cfn2dsl is a Ruby script that does something similar but instead outputs to Ruby code that the cfndsl gem uses and outputs a CloudFormation template when you feed it to a Ruby interpreter. That meant I had to create two custom Lambda deployment packages: One for cfn-flip and one for cfn2dsl.

Creating a deployment package for Python is a straightforward process. You can simply clone the cfn-flip repository and install the dependencies to your deployment package directory as follows:

```
pip install -r requirements.txt -t ~/your/deployment_package
```

On MacOS X, however, this doesn't work out of the box. You will need to create (or add to) setup.cfg in the cloned directory with the following content:

```
[install]
prefix=
```

By default you invoke the cfn-flip tool with some arguments to specify which file you want to convert. In this case, I wanted to have cfn-flip to use the content provided to the Lambda function. So essentially: whatever code the user submitted on the website. Looking at the source code there was just a single function in cfn-flip I needed, called **flip**.

My Lambda function then looked like this:

```
import json
import sys
import cStringIO
import urlparse
from cfn_flip import flip


def lambda_handler(event, context):
  data = urlparse.parse_qsl(event["body"])
  for k, v in data:
      if k == 'code':
          plaintext = v
      else:
          plaintext = ""

  stdout_ = sys.stdout
  stream = cStringIO.StringIO()
  sys.stdout = stream
  hndlr = sys.stdout
  hndlr.write(flip(
      plaintext
  ))
  sys.stdout = stdout_
  data = stream.getvalue()

  return {
    'statusCode': 200,
    'body': json.loads(json.dumps(data)),
    'headers': {
              'Content-Type': 'plain/text',
              'Access-Control-Allow-Origin': '*',
              'Cache-control': 'private, max-age=0, no-cache'
          },
      }
```

The event body expects the template's body being posted with code as key. The Lambda function then captures the output of cfn-flip's **flip** function, and finally returns the flipped version.

Porting Ruby's "cfn-dsl" was a different challenge altogether. I've used a portable version of Ruby 2.2.x called Traveling Ruby to prime my deployment package on a Linux environment. The cfn2dsl gem (rightfully) pinned to Ruby 2.3 as a minimum version, so I had to get creative.

I started with installing Ruby with an identical version of traveling ruby using rbenv. Then, I created a Gemfile with the dependencies of cfn2dsl, but without the versions pinned to a minimum version.

This allowed me to get all of cfndsl's gem dependencies that were compatible with Ruby 2.2. I then moved all the dependencies into the traveling ruby's gem directory and created a cfn2dsl_lib directory and stuck the cfn2dsl source code there. With that out-of-the-way I copied over the deployment package to my trusty Macbook and started to work on the Lambda function.

Now, AWS does not support Ruby natively, so the deployment package consists of a NodeJS function to invoke the Ruby script, and the Ruby script to transform the template into CfnDsl code.

The ruby code looked like this:

```
#!./bin/ruby

require "uri"
require_relative 'cfn2dsl_lib/cfn2dsl'

payload = URI.decode(JSON.parse(ARGV[0])["body"].tr('+', ' ')).
gsub(/^code\=/,'')
template = JSON.parse(payload)
cfndsl   = CloudFormation.new(template)
dsl = Render.new(cfndsl).cfn_to_cfndsl
```

```
dsl = "require 'cfndsl'\n\n" \
      "# Generated by https://cfnflip.com/\n" \
      "# The 'cfndsl' gem is required. Type `gem install
cfndsl` to install it.\n\n" \
      "#{dsl.gsub(/^CloudFormation\ do/, 'template = Cloud-
Formation do')}\n" \
      "puts template.to_json"

puts dsl
```

There are more eloquent ways to parse POST data, but I didn't include the required gems in the deployment package to avoid as much bloat as possible. Ruby can be notoriously slow. I broke down the cfn2dsl functionality to just rendering a template from a given string and outputting the result.

The NodeJS wrapper function looks like this:

```
const exec = require('child_process').exec;

exports.handler = function(event, context, callback) {
  const child = exec('./lambdaRuby.rb ' + '\'' + JSON.
  stringify(event) + '\'', function(error, stdout, stderr) {
      const response = {
        statusCode: 200,
        headers: {
          'Access-Control-Allow-Origin': '*',
          'Content-Type': 'text/html'
        },
        body: stdout
      };
      callback(null, response);
  });
  child.stdout.on('data', console.log);
  child.stderr.on('data', console.error);
}
```

The NodeJS function spawns the Ruby script as a child process, passing a JSON-encoded string as argument to the Ruby script, with a function that waits for the child process to terminate and fetch stdout. The output is then modified so it's ready to run out of the box and returned.

Last but not least, I needed to create the front-end which consists of a static HTML page and a Lambda function that returns the content:

```
'use strict';

var fs = require('fs');

module.exports.frontPage = (event, context, callback) => {
  var html = fs.readFileSync("index.html", "utf8");
  const response = {
    statusCode: 200,
    headers: {
      'Access-Control-Allow-Origin': '*',
      'Content-Type': 'text/html',
    },
    body: html
  };

  callback(null, response);
};
```

The Lambda function reads the contents of index.html and returns it.

Anatomy of API Gateway

With the Lambda functions in place, it is time to build an API gateway that terminates different end-points to their corresponding Lambda functions. We have a Lambda for the main page, and two Lambda's to convert the submitted templates to a different format. I will highlight some parts of the CloudFormation template that are key. If you're interested in the full template you can find it on GitHub.

First we create our API Gateway:

```
  ApiGatewayRestApi:
    Type: AWS::ApiGateway::RestApi
    Properties:
      Name: cfnflip
```

We create our /cfn2dsl/ endpoint as follows:

```
  ApiGatewayResourceRuby:
    Type: AWS::ApiGateway::Resource
    Properties:
      ParentId: !GetAtt 'ApiGatewayRestApi.RootResourceId'
      PathPart: cfn2dsl
      RestApiId: !Ref 'ApiGatewayRestApi'
```

For each Lambda I've created a log group to log to a location that's predictable and consistent. In this case the log group name is identical to the name of the Lambda function. In case of the Ruby Lambda function the log group resource looks like this:

```
  CfnFlipRubyLogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      RetentionInDays: 7
      LogGroupName: /aws/lambda/cfn2dsl
```

The LogGroupName determines where in CloudWatch you'll be able to find back the log files, and we don't want to keep them forever so we throw away the logs after 7 days. Remember the LogGroupName. We'll get back to it when creating the IAM AssumeRole policy.

To deploy the Lambda function we need to create an AWS::Lambda::Function resource:

```
  CfnFlipRubyLambdaFunction:
    Type: AWS::Lambda::Function
    Properties:
      Code:
        S3Bucket: !ImportValue 'CfnFlipCloudformationBucket'
        S3Key: cfn2dslb538066ac5e095501bfe89eccc015a7f.zip
      Handler: ruby.handler
      FunctionName: cfn2dsl
      MemorySize: 256
      Role: !GetAtt 'IamRoleLambdaExecution.Arn'
      Runtime: nodejs6.10
      Timeout: 10
    DependsOn:
      - IamRoleLambdaExecution
      - CfnFlipRubyLogGroup
```

As you can see we import the bucket from the dependency stack and pass it to S3Bucket. In our deployment package we've created ruby.js, which we use as the function's handler. MemorySize is all about balance. The higher the memory size, the more CPU the function has access to, so it's all about balance between speed and cost. For our Ruby lambda function 256 MB of memory provided the best balance. When sizing Lambda functions you have to keep in mind that smaller sizing might mean more costs per invocation, and that sizing high might mean the increased speed doesn't justify the additional costs.

Then, finally, we need to create a POST method for our endpoint:

```
ApiGatewayMethodRubyPost:
  Type: AWS::ApiGateway::Method
  Properties:
    HttpMethod: POST
    RequestParameters: {}
    ResourceId: !Ref 'ApiGatewayResourceRuby'
    RestApiId: !Ref 'ApiGatewayRestApi'
    AuthorizationType: NONE
    Integration:
      RequestTemplates:
        application/x-www-form-urlencoded: "#set($allParams
        = $input.params())\n\
          {\n  \"params\" : {\n     #foreach($type in
          $allParams.keySet())\n     #set($params\
          \ = $allParams.get($type))\n     \"$type\" :
          {\n       #foreach($paramName\
          \ in $params.keySet())\n       \"$paramName\" :
          \"$util.escapeJavaScript($params.get($paramName))\"\
          \n       #if($foreach.hasNext),#end\n       #end\n
                   }\n     #if($foreach.hasNext),#end\n\
          \    #end\n  }\n}\n"
      IntegrationHttpMethod: POST
      Type: AWS_PROXY
      Uri: !Join
        - ''
        - - 'arn:aws:apigateway:'
          - !Ref 'AWS::Region'
          - :lambda:path/2015-03-31/functions/
          - !GetAtt 'CfnFlipRubyLambdaFunction.Arn'
          - /invocations
    MethodResponses: []
```

The RequestTemplate in the above snippet transforms the request into an event body that we can use in the function, and then we pass it on to the Lambda function.

So did you remember to LogGroupName ? It will come into play now. As you can see in the Lambda function snippet, there is a reference to the IamRoleLambdaExecution role. This role provides the Lambda function with relevant permissions to perform actions on your behalf. You want to limit these privileges to the least amount required for the function to work. Our Lambda's fortunately don't require a lot of permissions. It doesn't interact with any AWS service nor does it require any VPC to talk to.
So what does it need? It creates CloudWatch log streams and write to them! So let's have a look:

```
IamRoleLambdaExecution:
  Type: AWS::IAM::Role
  Properties:
    AssumeRolePolicyDocument:
      Statement:
        - Action:
            - sts:AssumeRole
          Effect: Allow
          Principal:
            Service:
              - lambda.amazonaws.com
      Version: '2012-10-17'
    Policies:
      - PolicyDocument:
          Statement:
            - Action:
                - logs:CreateLogStream
              Effect: Allow
              Resource:
                - !Sub 'arn:aws:logs:${AWS::Region}:
                  ${AWS::AccountId}:log-group:/aws/lambda/
                  cfnflip:*'
                - !Sub 'arn:aws:logs:${AWS::Region}:
                  ${AWS::AccountId}:log-group:/aws/lambda/
                  pyflip:*'
                - !Sub 'arn:aws:logs:${AWS::Region}:
                  ${AWS::AccountId}:log-group:/aws/lambda/
                  cfn2dsl:*'
            - Action:
                - logs:PutLogEvents
              Effect: Allow
              Resource:
                - !Sub 'arn:aws:logs:${AWS::Region}:
                  ${AWS::AccountId}:log-group:/aws/lambda/
                  cfnflip:*:*'
                - !Sub 'arn:aws:logs:${AWS::Region}:
                  ${AWS::AccountId}:log-group:/aws/lambda/
                  pyflip:*:*'
                - !Sub 'arn:aws:logs:${AWS::Region}:
                  ${AWS::AccountId}:log-group:/aws/lambda/
                  cfn2dsl:*:*'
          Version: '2012-10-17'
        PolicyName: !Join
          - '-'
          - - cfnflip
            - lambda
    Path: /
    RoleName: !Join
      - '-'
      - - cfnflip
        - eu-west-1
        - lambdaRole
```

We allow the Lambda function to assume this role with all the permissions listed in the PolicyDocument. We grant the CreateLogStream and the PutLogEvents privilege to the LogGroups we've created before. This is all our functions require.

Epilogue

It's 2:27 AM and I'm typing this very epilogue. It took me longer to write this blog post than it took me to create the MVP for cfnflip.com. It was great fun to make and I'm already brewing some new ideas. My esteemed colleague Mark is toying with the thought of whipping up some Terraform transformer. If you have any cool ideas do let us know, or better yet: Fork the project and submit a PR for your improvements.

Dennis Vink
AWS Consultant
dennisvink@binx.io

Keeping Your AMIs up to Date in CloudFormation

Referencing virtual machines images in your CloudFormation template is hard. You have to lookup the image you want to use and copy its non-descriptive id (ami-xxxxxxx) into the template. If you deploy to multiple regions, you have to lookup and add these ids too. Whenever a new version of the virtual machine image is available, you have to repeat the whole process. With our Custom CloudFormation Provider and maintenance utility, referencing virtual machine images becomes both easy to read and maintain.

Author Mark van Holsteijn

To ease the specification and maintenance of virtual machine image ids in CloudFormation, we created a Custom CloudFormation Resource **`Custom::AMI`** and a maintainance utility called **`aws-cfn-update`**.

The Custom Resource allows you to specify a virtual machine image by name. The utility **`aws-cfn-update`** provides the ability to update these Custom Resource definitions in your templates from the command line.

Specifying an AMI by name

To specify an AMI by name, add a Custom::AMI resource to your template:

```
AMI:
    Type: Custom::AMI
    Properties:
      Filters:
        name: 'amzn-ami-2017.09.a-amazon-ecs-optimized'
      ServiceToken: !Sub 'arn:aws:lambda:${AWS::
      Region}:${AWS::AccountId}:function:binxio-cfn-ami-
      provider'
```

The **`Filters`** property allows you to specify which AMI you want. The name of the image is sufficient, but if you want you can go wild and specify any of the parameters defined in the EC2 describe-images API call.

Once you have added the custom AMI resource definition, you can get the id as follows:

Instance:

```
    Type: AWS:EC2::Instance
    EC2Instance:
        Type: AWS::EC2::Instance
        Properties:
            ImageId: !Ref AMI
```

Keeping your AMI up to date

To keep the AMI up to date, you can use the utility aws-cfn-update:

```
aws-cfn-update latest-ami --ami-name-pattern 'amzn-ami-
2017.09.a-amazon-ecs-optimized'
```

The utility takes an AMI name pattern to search for the latest AMI using the describe-images API. The utility will update the **`name`** property of the filter if a newer version exists.

Do you want to try it? Than install the custom resource provider and the utility.

Installing the custom resource provider

To install this custom resource provider, type:

```
aws cloudformation create-stack \
    --capabilities CAPABILITY_IAM \
    --stack-name cfn-ami-provider \
    --template-body file://cloudformation/cfn-ami-provider.json

aws cloudformation wait stack-create-complete  --stack-name
cfn-ami-provider
```

This CloudFormation template will use our pre-packaged provider from **`s3://binxio-public-${AWS_REGION}/lambdas/cfn-ami-provider-latest.zip`**.

Installing the aws-cfn-updae utility

To install the **`aws-cfn-update`** utility, type:

```
pip install aws-cfn-update
```

Demo

To demonstrate the use of the Custom Resource, type:

```
aws cloudformation create-stack --stack-name cfn-ami-
provider-demo \
    --template-body file://cloudformation/demo-stack.json
aws cloudformation wait stack-create-complete  --stack-name
cfn-ami-provider-demo
```

and update the template, by typing:

```
aws-cfn-update latest-ami --ami-name-pattern 'amzn-ami-
2017.09.a-amazon-ecs-optimized ./cloudformation
```

Checkout the updated Custom::AMI resource definition (git diff) and update the template:

```
aws cloudformation update-stack --stack-name cfn-ami-
provider-demo \
        --template-body file://cloudformation/demo-stack.json
aws cloudformation wait stack-update-complete  --stack-name
cfn-ami-provider-demo
```

Conclusion

With this custom CloudFormation Provider and update utility you can declare and maintain an AMI by name, which will ease the maintainability of your CloudFormation templates.

If you have any questions, do not hesitate to contact me.

Dennis Vriend
Cloud Consultant
dennisvriend@binx.io

A Fast and Easy Way to Deploy Scala Based Lambdas to AWS Using Sbt

The development cycle of AWS lambdas in Scala is hell. There are so many packaging and configuration steps that it destroys my productivity. To solve this, we created a Scala Build Tool Plugin called sbt-sam. The plugin makes deploying serverless applications as easy as compiling your code. In this blog we will show you how easy it is to create a REST service using Scala, AWS API Gateway and AWS Lambda.

We assume the following:
- you are using OSX
- you have SBT installed
- you have an AWS account
- the AWS account has enough privileges to deploy serverless applications

You can create a serverless application in Scala using AWS Lambda in 3 easy steps:
1. Create a SBT project
2. Deploy the application
3. Call the application
4. Create a SBT project

To create a new serverless application project, type:
```
sbt new dnvriend/sam-seed.g8
```

The project asks some questions, press enter to accept the defaults:
```
name [sam-seed]:
package [com.github.dnvriend]:
organization [com.github.dnvriend]:
author _ name [dnvriend]:
stage [dev]:

Template applied in ./sam-seed
```

To sbt in the created project type:
```
$ cd sam-seed
$ sbt
```

The plugin adds the following commands to SBT:
- samInfo - to get information from your serverless application
- samValidate - to view the generated CloudFormation template
- samDeploy - to deploy the serverless application
- samRemove - to remove the serverless application

The template lambda implementation is show below:
```
package com.github.dnvriend

import com.github.dnvriend.lambda.annotation.HttpHandler
import com.github.dnvriend.lambda._
import play.api.libs.json._

@HttpHandler(path = "/hello", method = "get")
 class HelloLambda extends ApiGatewayHandler {
    override def handle(
      request: HttpRequest,
      ctx: SamContext
    ): HttpResponse = {
        HttpResponse
          .ok
          .withBody(Json.toJson("Hello world"))
    }
}
```

The HelloLambda class extends ApiGatewayHandler, which makes it an AWS Lambda. The lambda is annotated with HttpHandler with the path '/hello' and the method 'get'. This means the lambda is accessible via API Gateway.

Deploy the application

Deploy the application by typing `samDeploy`:

```
sbt:sam-seed> samDeploy
[info] Creating cloud formation stack
[info] CREATE _ IN _ PROGRESS - AWS::S3::Bucket -
SbtSamDeploymentBucket - CREATE _ IN _ PROGRESS -
[info] CREATE _ IN _ PROGRESS - AWS::S3::Bucket - SbtSamDeploy-
mentBucket - CREATE _ IN _ PROGRESS - Resource creation
Initiated
...
[success] Total time: 102 s, completed Apr 11, 2018 7:04:00 PM
```

After the service is deployed, use samInfo to get the URL to call the service:

```
sbt:sam-seed> samInfo
[info] ====================
[info] Name: com-github-dnvriend-dnv-sam-seed
[info] Description: com-github-dnvriend-dnv-sam-seed
[info] Status: UPDATE _ COMPLETE
...
[info] Endpoints:
[info] GET - https://uu0vpzo9f0.execute-api.eu-west-1.amazonaws.
com/Prod/hello
```

Call the application

Call the endpoint by typing:

```
$ curl https://uu0vpzo9f0.execute-api.eu-west-1.amazonaws.com/
Prod/hello
"Hello world"
```

Change the text 'Hello World' to 'Goodbye World' and type samDeploy. The new code will be deployed to AWS and call the same endpoint again:

```
$ curl https://uu0vpzo9f0.execute-api.eu-west-1.amazonaws.com/
Prod/hello
"Goodbye World"
```

Conclusion

With the SAM plugin for SBT, developing serverless applications is as easy as compiling code. A quick way to create serverless applications is to make use of template projects. But it does not end here. The plugin support many more AWS services and features. We will demonstrate how to use these in future blogs.

Next time we'll look how to secure the web service using Amazon Cognito User Pools. We will create a Scala application that authenticates with Cognito and calls the web service.

Quby

Customer Story Quby

Quby is the smart energy platform that puts people in charge of their homes

In 2017, binx.io migrated the 10-year old backend application for Quby's smart thermostat from their existing data-center to the Cloud.

The main goal of the migration, was to be enabled to sell their flagship product to multiple energy companies across Europe and increase the number of users from 400k to 20 million users. After the migration, they were able to add new customers in a matter of hours, improved the stability of their backend platform significantly and obtained a fully automated ci/cd pipeline for all of their applications.

Binx.io designed and implemented the backend platform, modernized the application packaging and deployment using Docker and trained their current IT personnel to be able to continue to improve the platform.

Customer Story COIN

COIN is a collaboration between Dutch providers of electronic communication services and networks

They provide joint and reliable services within the telecom sector and offers services that help telecom providers to work together more efficiently.

In 2017, binx.io designed and implemented a cloud platform for all of the services of COIN in AWS. Before that time, all of the IT systems were managed and operated by third party suppliers. As a result, the exploitation costs of their IT was high and making changes was slow and expensive.

After the migration, COIN is able to make changes to their systems in house using a Agile development process. The platform itself is high-available and self-healing, increasing the uptime and response time of the services without manual intervention.

Binx.io designed and implemented the cloud platform, and helped to form an in-house software delivery team, supported by a fully automated continuous software delivery pipeline.

vereniging COIN

Generate Temporary AWS Credentials

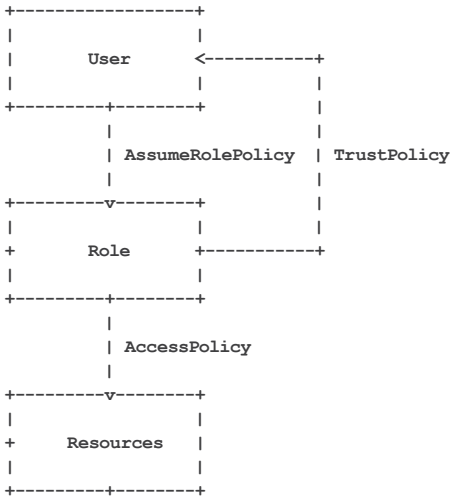
For our product Instruqt, we're building challenges to learn AWS technology. Users of instruqt need to have temporary access to create, update and destroy their resources in AWS. We would like to give the users the experience of having their own AWS account. Access keys to use in the CLI or programmatic access, and the Management Console in the browser. This blog post describes how to prepare an AWS account for this usage, and how to generate and use temporary access keys, and a magic link to the management console.

Prerequisites

- **AWS Account.** You need to have an AWS Account, with a user and access keys with permissions to list, read, create, update and delete IAM groups, users, roles and policies.
- **Terraform.** You need to have installed Terraform.
- **Python.** Ensure you have a recent python version with pip. On a mac I recommend this setup.
- **AWS CLI.** Install the AWS CLI with pip. (`pip install awscli --upgrade --user`)
- **Keybase.** Install and create a Keybase account (https://keybase.io/)

Roles and Federation

Besides common entities like users and groups, AWS provides "roles". A role in AWS could for example be assigned to a server, which allows software running on the server to access AWS resources. A role can also be assumed by a user, giving him access to the resources. Also known as "federation". A role contains two types of policies. One policy which describes the type of the service allowed to assume the role (an ec2 instance or an AWS account with users). The other policy describes the permission level to the specified resources.



Setup

To setup the user, role, policies etc, you could use CloudFormation, Terraform, the CLI or the Management Console. In this example I've used Terraform, but you could easily use this as documentation to do it manually, or using CloudFormation.

Create **setup.tf** using the following source. You won't have to change anything, but it's recommended to read and search in the terraform documention what exactly happens.

```
# CONFIGURATION AND PARAMETERS

variable "aws" {
  description = "Enter the aws profile to deploy."
}

variable "keybase" {
  description = "Enter the keybase profile to encrypt the secret_key (to decrypt: terraform output secret_key | base64 --decode | keybase pgp decrypt)"
}

variable "region" {
  default = "eu-west-1"
}

provider "aws" {
  profile    = "${var.aws}"
  region     = "${var.region}"
}

data "aws_caller_identity" "current" {}

# RESOURCES

resource "aws_iam_user" "instruqt" {
  name = "instruqt"
}

resource "aws_iam_access_key" "instruqt" {
  user = "${aws_iam_user.instruqt.name}"
  pgp_key = "keybase:${var.keybase}"
}

resource "aws_iam_user_policy" "instruqt_assume_role" {
  name = "test"
  user = "${aws_iam_user.instruqt.name}"

  policy = <<EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "sts:Assume*"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
EOF
}
```

```
resource "aws_iam_role" "S3AccessRole" {
  name = "InstruqtS3Access"
  assume_role_policy = <<EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": "sts:AssumeRole",
      "Principal": {
        "AWS": "arn:aws:iam::${data.aws_caller_identity.cur-
rent.account_id}:root"
      },
      "Effect": "Allow",
      "Sid": ""
    }
  ]
}
EOF
}

resource "aws_iam_role_policy" "S3AccessPolicy" {
  name = "InstruqtS3AccessPolicy"
  role = "${aws_iam_role.S3AccessRole.id}"
  policy = <<EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "s3:*"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
EOF
}

# OUTPUT

output "role_arn" {
  value = "${aws_iam_role.S3AccessRole.arn}"
}
output "access_key" {
  value = "${aws_iam_access_key.instruqt.id}"
}
output "secret_key" {
  value = "${aws_iam_access_key.instruqt.encrypted_secret}"
}
```

Deploy the stack using the following commands. Select one of the profiles configured in **~/.aws/credentials**. Keep the profile name and your keybase username ready, because the apply command will ask you to enter these values.

```
terraform init
terraform apply
```

When Terraform has completed, the output contains access keys for the next steps. The secret key is encrypted with keybase. Decrypt the generated secret key using the next command:

```
terraform output secret_key | base64 --decode | keybase
pgp decrypt
```

Run the python generate_keys.py

First install dependencies with for example pip, using the next example. When you get errors of missing dependencies, install those as well.

```
pip install requests boto3
```

Create a file **generate_keys.py** with the following source:

```
#!/usr/bin/env python
import os
import sys
import boto3
import json
import requests
import argparse
import ConfigParser
from botocore.exceptions import ClientError

def get_credentials_for_role(role_arn, session_name):
    sts = boto3.client('sts')
    try:
        response = sts.assume_role(RoleArn=role_arn,
                                   RoleSessionName=
                                   session_name)
        return response['Credentials']
    except ClientError as e:
        sys.stderr.write('ERROR: %s\n' % e.response['Error']
['Message'])
        sys.exit(1)

def write_credentials(profile, credentials):
    filename = os.path.expanduser('~/.aws/credentials')
    dirname = os.path.dirname(filename)

    if not os.path.exists(dirname):
        os.makedirs(dirname)

    config = ConfigParser.ConfigParser()
    config.read(filename)
    if not config.has_section(profile):
        config.add_section(profile)
    config.set(profile, 'aws_access_key_id', credentials
    ['AccessKeyId'])
    config.set(profile, 'aws_secret_access_key',
    credentials['SecretAccessKey'])
    config.set(profile, 'aws_session_token', credentials
    ['SessionToken'])
    with open(filename, 'w') as fp:
        config.write(fp)

def generate_console_link(credentials):
    session = json.dumps({'sessionId': credentials['Access-
KeyId'],
                          'sessionKey': credentials
                          ['SecretAccessKey'],
                          'sessionToken': credentials
                          ['SessionToken']})

    r = requests.get("https://signin.aws.amazon.com/federation",
                     params={'Action': 'getSigninToken',
                             'SessionDuration': 43200,
                             'Session': session})
    signin_token = r.json()

    console = requests.Request('GET',
                               'https://signin.aws.amazon.com/
                               federation',
                               params={'Action': 'login',
                                       'Issuer': 'Instruqt',
                                       'Destination': 'https://
                                       console.aws.amazon.com/',
                                       'SigninToken': signin_
                                       token['SigninToken']})
    prepared_link = console.prepare()
    return prepared_link.url

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='generate
    keys')
    parser.add_argument("--output", "-o", required=False,
                        dest="output", help="output format",
                        metavar="STRING",
                        default="json", choices=['link', 'json',
                        'write'])
    parser.add_argument("--role-arn", "-r", required=True,
                        dest="role_arn", help="to assume",
                        metavar="STRING")
    parser.add_argument("--session-name", "-s",
    required=True,
                        dest="session_name", help="to use",
                        metavar="STRING")

    options = parser.parse_args()
    credentials = get_credentials_for_role
    (options.role_arn,
                                           options.session_
                                           name)
```

```
if options.output == 'link':
    print generate_console_link(credentials)
elif options.output == 'write':
    write_credentials(options.session_name,
    credentials)
elif options.output == 'json':
    print(json.dumps({'AccessKeyId': credentials
    ['AccessKeyId'],
                      'SecretAccessKey': credentials
                      ['SecretAccessKey'],
                      'SessionToken': credentials
                      ['SessionToken'],
                      'ConsoleMagicLink': generate_
                      console_link(credentials)}))
```

The ./generate_keys.py script requires 3 parameters:

- **session-name.** The session name is an unique ID of the user who is going to use the temporary credentials. (Example: martijn@binx.io)
- **role-arn.** The role arn is part of the output of the terraform script. (Example: arn:aws:iam::AWS_ACCOUNT_ID:role/InstruqtS3Access). You can copy this from the output of terraform.
- **output.** This could only contain: json | write | link, default output is json.

generate_keys.py uses Boto (AWS SDK for Python). It will use environment variables for access keys. Use the first example or copy the access_key and role_arn output from terraform, the decrypted secret_key and replace the example variables in the second example.

```
AWS_ACCESS_KEY_ID=$(terraform output access_key) \
AWS_SECRET_ACCESS_KEY=$(terraform output secret_key |
base64 --decode | keybase pgp decrypt) \
python ./generate_keys.py --session-name identified@domain.
ext \
                          --role-arn $(terraform output role_
                          arn) \
                          --output json
AWS_ACCESS_KEY_ID=AKIA34K435KLR12KDT345 \
AWS_SECRET_ACCESS_KEY=lk45hJSFkl35ADfsdDFtkl34fFADFhlktj-
rfaewr \
python ./generate_keys.py --session-name identified@domain.
ext \
                          --role-arn arn:aws:iam::AWS_
                          ACCOUNT_ID:role/InstruqtS3Access \
                          --output json
```

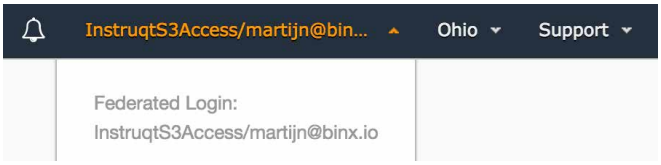
Martijn van Dongen
Chief AWS
martijnvandongen@binx.io

If you get an error with at the end of the stack trace the message: "The security token included in the request is invalid." It's probably because you didn't replace the values of AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY.

Use the generated temporary keys

With a browser

Open a browser and copy the "console_access" magic link. You're automatically logged on.



With the CLI

There are several ways to use the temporary credentials. To add the credentials once and easily use it in next commands, you create a new profile in **~/.aws/credentials**. With --output write, the section is directly written into the credentials file and ready to be used.

```
aws s3 ls --profile tmpinstruqt
```

Programmatic access

And you can of course use these credentials for programmatic access. This is not the recommended way of adding secrets in your code, but just for this example hard coded.

```
import boto3
client = boto3.resource(
    's3',
    aws_access_key_id='ASIAL34H2K3423KL4JLKJLKJ',
    aws_secret_access_key='e0hvLM234LKJ23KDAdsf
    23DFAXiBrNu8Ht',
    aws_session_token='F4adsJL2sdafK3J42K3LJ4erg2K3J4....',
)
for bucket in client.buckets.all():
    print(bucket.name)
```

Final words

Now everything is setup and tested, just try to make it awesome. Give the temporary users more permissions to do other stuff than S3, generate and use the new keys in the CLI, scripts or using the Management Console.

To clean up, just remove the temporary credentials from your **~/.aws/credentials**, and **clear** the terminal. Run **terraform destroy** to remove the AWS setup.

You can download the source from github: https://github.com/binxio/generate-temp-aws-credentials

Feel free to contact me if you have any questions. Your feedback is appreciated, and will be used to improve this blog post and future posts.

NB. Big thanks to my colleague Mark van Holsteijn.
The python code used in the first version was copied from AWS Documentation. It wasn't that clean. So we refactored it to the script now used in the blog post. It also gave us inspiration for new blog posts, so stay tuned for updates and related stuff.

Customer Story Royal FloraHolland

For over one hundred years, Royal FloraHolland, its growers, and customers have been making the world a healthier and more beautiful place with flowers and plants

Every day, streamlined logistics and smart digital services ensure that 400,000 species of flowers and plants arrive at their worldwide destinations.

In 2017, Royal FloraHolland launched the Digital Greenhouse, a global trade platform for floriculture. In addition to digitizing their existing services, it also worked with Xebia Group to develop new and innovative solutions to help growers and buyers find each other more easily. These solutions include deep learning applications, up-to-date reporting, predictions and smart recommenders.

Floriculture auctions are being digitized and attract a more global audience. To remain relevant for growers and traders, it is essential for Royal FloraHolland to provide relevant services.Together with Xebia, Royal FloraHolland is digitizing their existing services, and develops new and innovative solutions to help growers and buyers find each other more easily.

Royal Flora Holland

Deploying Aws Ses Access Key and Smtp Password to the Parameter Store Using AWS Cloudformation

In AWS CloudFormation there is no way to generate the SMTP password of an AWS access key. As a result, the application always has to do the calculation and transform the secret key into an SMTP password.

Author Mark van Holsteijn

With This custom CloudFormation provider, we put an end to that. You can create an access key and SMTP password and automatically store the credentials in the AWS Parameter Store. This means that you can create the email infrastructure and provision SMTP credentials to applications that need to send email through Amazon Simple Email Service in a safe and controlled manner.

How does it work?

It is quite easy: you add the CloudFormation resource Custom::AccessKey, as follows:

```
Resources:
  AccessKey:
    Type: Custom::AccessKey
    Properties:
      Description: sample user credential
      UserName: '<UserName>'
      ParameterPath: '<Parameter Path>'
      ServiceToken: !Sub 'arn:aws:lambda:
      ${AWS::Region}:${AWS::AccountId}:function:
      binxio-cfn-secret-provider'
```

The access key id, access secret and the SMTP password are stored in the parameter store under the paths **<ParameterPath>/aws_access_key_id**, **<ParameterPath>/aws_access_secret_key** and **<ParameterPath>/smtp_password respectively**.

Properties

You can specify the following properties:

- **UserName** - to create an access key for.
- **ParameterPath** - into the parameter store to store the credentials
- **Serial** - to force the access key to be recycled
- **Status** - Active or Inactive
- **ReturnSecret** - returns access id and access secret as attribute
- **ReturnPassword** - returns access id and SMTP password as attribute
- **NoEcho** - indicate whether output of the return values is replaced by *****, default True.

Return values

With 'Fn::GetAtt' the following values are available:

- **SMTPPassword** - the SMTP password based for the access key (if ReturnPassword is true).
- **AccessSecretKey** - the secret part of the access key (if ReturnSecret is true).

For more information about using Fn::GetAtt, see Fn::GetAtt.

Installation

To install this Custom Resource, type:

```
git checkout https://github.com/binxio/cfn-secret-provider
cd cfn-secret-provider

aws cloudformation create-stack \
        --capabilities CAPABILITY_IAM \
        --stack-name cfn-secret-provider \
        --template-body \
        file://cloudformation/cfn-custom-resource-provider.json

aws cloudformation wait stack-create-complete  \
        --stack-name cfn-secret-provider
```

This CloudFormation template will use our pre-packaged provider from:

```
s3://binxio-public-{{AWS::Region}}/lambdas/cfn-secret-
provider-latest.zip
```

Demo

To install the simple sample from this blog post, type:

```
aws cloudformation create-stack \
        --stack-name cfn-secret-provider-demo \
        --template-body file://cloudformation/demo-stack.json

aws cloudformation wait stack-create-complete  \
        --stack-name cfn-secret-provider-demo
```

to validate the result, type:

```
aws ssm get-parameters-by-path --path /iam-users --recursive
--with-decryption
```

Conclusion

By using the Custom CloudFormation Secret provider you can create an IAM Access Key and the derived SMTP password and stored in the parameter store where it is encrypted and access can be audited and controlled.

binx.io
proudly part of Xebia Group

Laapersveld 27 / 1213 VB Hilversum
Wibautstraat 202 / 1091 GS Amsterdam
+31 35 538 1921 / @binxio / binx.io