Infrastructure as Code

Comparing CloudFormation, Terraform, and CDK

Keith Gregory
AWS Practice Lead
Chariot Solutions

The Players

CloudFormation

Official AWS tool for declarative deployments

Templates defined in JSON or YAML, used to create *Stacks*

- Can create multiple stacks from same template, using parameters to configure

All resources must be declared individually

- Example: if you need 14 users, you'll have 14 `AWS::IAM::User` resource declarations
- Maximum number of resources per stack: 200

Modularity via nested templates/stacks

Can export resource references for use by another stack

As-of November 2019, can import existing resources into stack

Terraform

Multi-provider declarative infrastructure tool, produced by HashiCorp

Configurations defined using HCL: HashiCorp Configuration Language

Deployments tracked using “tfstate”

- Must be preserved: checked-in to source control, stored on S3, or in HashiCorp repository
- Warning: secrets stored in plaintext!

Can create multiple resources from a single declaration

- Example: to create 14 users, first create a variable that lists those users’ names, then create an `aws_iam_user` resource based on the size of this list

Can create *modules* for reusability

Can import existing AWS resources to bring them under Terraform control

Cloud Development Kit (CDK)

Open-source tool to generate/deploy CloudFormation templates

- Generated templates contain same content as hand-written

You write an *Application*, which manages one or more *Stacks*

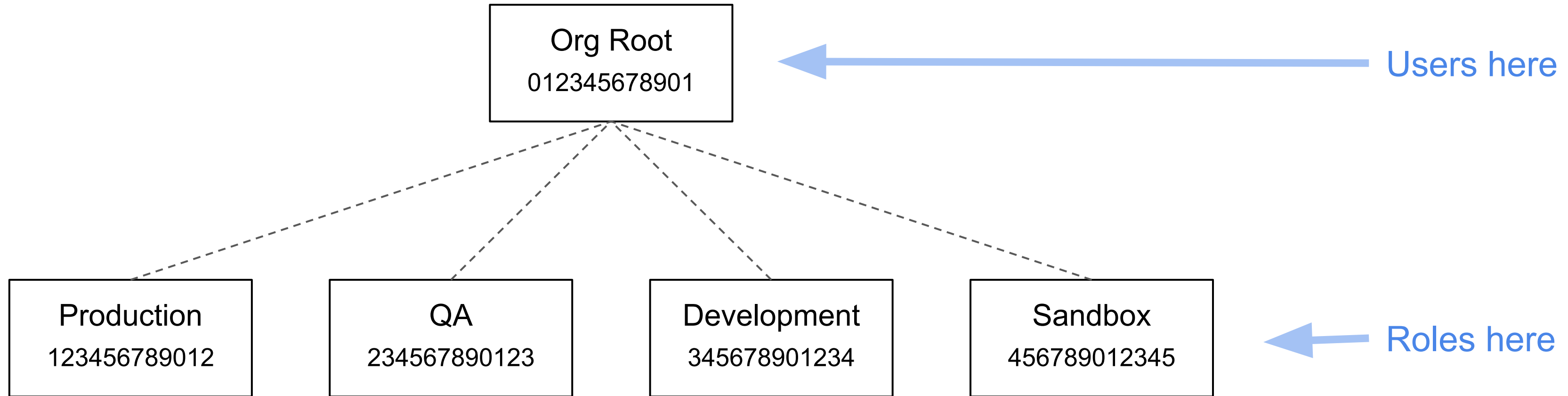
- Default language: TypeScript
- Also supports: JavaScript, Python, Java, and .Net

Constructs provide reusability

- Low-level constructs represent CloudFormation resources (eg: users)
- Higher-level constructs can create multiple resources, provide default values
- CDK includes a library of constructs, or you can create your own

Task 1: Manage Users, Groups, Roles

Cross-Account Permissions Management



Roles in each child grant varying permissions

Users defined in parent, belong to Groups

Groups grant permission to assume roles

Group Permissions Policy

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "sts:AssumeRole",
            "Resource": "arn:aws:iam::123456789012:role/ExampleProductionRole"
        }
    ]
}
```

Attached to a group, applied to all users in group

Members of group allowed to assume one or more *specific* roles

- Can specify multiple roles, corresponding to different applications/services

Role Trust Policy

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::012345678901:root"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Each assumable role must include this trust policy

Anyone in account 012345678901 that *can* assume the role *may* do so

Teh Codez

CloudFormation: Basic Template

```
AWSTemplateFormatVersion:                    "2010-09-09"
Description:                                 "Describe the stack"

Parameters:

  # parameters allow you to provide configuration

Resources:

  # resources are things that are created in your account

Outputs:

  # outputs let you expose attributes of created resources (eg, RDS hostname)
```

CloudFormation: Creating Users and Groups

```
Resources:

  User1:
    Type:                           "AWS::IAM::User"
    Properties:
      UserName:                     "user1"
      ManagedPolicyArns:            [ !Sub "arn:aws:iam::${AWS::AccountId}:policy/BasicUserPolicy" ]
      Groups:
        -                           !Ref Group1
        -                           !Ref Group2

  Group1:
    Type:                           "AWS::IAM::Group"
    Properties:
      GroupName:                    "group1"

  Group2:
    Type:                           "AWS::IAM::Group"
    Properties:
      GroupName:                    "group2"
```

This policy was created outside of the stack

These references establish execution graph

CloudFormation: Assigning Roles to Groups

```
Group1Policy:
  Type:                              "AWS::IAM::Policy"
  Properties:
    Groups:                          [ !Ref Group1 ]
    PolicyName:                      "AllowRoleAssumption"
    PolicyDocument:
      Version:                       "2012-10-17"
      Statement:
        -
          Effect:                    "Allow"
          Action:                    [ "sts:AssumeRole" ]
          Resource:
           -                         !Sub "arn:aws:iam::${DevAccountId}:role/FooAppDeveloperRole"
           -                         !Sub "arn:aws:iam::${ProdAccountId}:role/FooAppReadOnlyRole"
```

CloudFormation: Parameterizing Account IDs

```
Parameters:

  ProdAccountId:
    Description:                    "Symbolic name for the production account ID"
    Type:                           "String"
    Default:                        "123456789012"

  DevAccountId:
    Description:                    "Symbolic name for the development account ID"
    Type:                           "String"
    Default:                        "345678901234"
```

Terraform: Creating Users

```
provider "aws" {}
```

Terraform can support multiple cloud providers

```
variable "users" {
    type = list
    default = [ "user1", "user2", "user3" ]
}
```

Variables are like CloudFormation parameters

Identifies the resource type

Name for specific resource definition

```
resource "aws_iam_user" "users" {
  count = length(var.users)
  name  = "${var.users[count.index]}"
}
```

This determines how many users we'll create

This retrieves a different username for each item

Terraform: Assigning Basic User Policy

```
data "aws_caller_identity" "current" {}

resource "aws_iam_user_policy_attachment" "base_user_policy_attachment" {
  count      = length(var.users)
  user       = "${var.users[count.index]}"
  policy_arn = "arn:aws:iam::${data.aws_caller_identity.current.account_id}:policy/BasicUserPolicy"
}
```

Data objects can retrieve information from AWS

One policy attachment per user

Identifies the data object

Identifies the object instance

An object has multiple named attributes

Terraform: Assigning Users to Groups

```
variable "group_members" {
    type = map(list(string))
    default = {
      "user1"  = [ "group1", "group2" ],
      "user2"  = [ "group1" ],
      "user3"  = [ "group2" ]
    }
}


resource "aws_iam_user_group_membership" "group-membership" {
  count = length(var.users)
  user  = "${var.users[count.index]}"
  groups = "${var.group_members[var.users[count.index]]}"
}
```

CDK: Getting Started

Each CDK project lives in its own directory, which must be initialized

```
cdk init app --language=typescript
```

This creates the directory structure, downloads modules needed for every project

You also need to install modules for each service that you use

```
npm install @aws-cdk/aws-iam
```

It’s a program, so you need to build it before use

```
npm run build
```

You can then either produce a CloudFormation template or deploy directly

```
cdk synth > template.json
```

```
cdk deploy
```

CDK: Things that go in Source Control

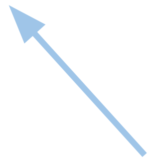
`package.json`	NPM configuration file that defines build commands and dependencies.
`package-lock.json`	NPM configuration file that identifies specific dependency versions, for all transitive dependencies.
`tsconfig.json`	Configuration file for the TypeScript compiler.
`cdk.json`	Project configuration file. At a minimum contains the command used to build and run the project; can contain runtime config.
`cdk.context.json`	Created by CDK to store runtime context values (eg, VPC ID).
`bin/main.ts`	The CDK “application” (may have a different name).
`lib/*`	User code for stacks and other constructs.

CDK: main.ts

```
#!/usr/bin/env node
import 'source-map-support/register';
import cdk = require('@aws-cdk/core');

import { UsersAndGroupsStack } from '../lib/stack';

const app = new cdk.App();
new UsersAndGroupsStack(app, 'UsersAndGroupsStack');
```



One application can build many stacks

CDK: stack.ts

```
import cdk = require('@aws-cdk/core');
import iam = require('@aws-cdk/aws-iam');

const userNames : string[] = [ "user1", "user2", "user3" ]

export class UsersAndGroupsStack extends cdk.Stack {

  constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    let stack = this

    // see next slide
  }
}
```

Stacks are a construct that runs within another construct (the app)

You need to pass a reference to the stack to set scope for other constructs, but it's JavaScript, so "this" changes all the time

You define all of the resources within the stack's constructor; this isn't particularly object-oriented, and limits refactoring

CDK: stack.ts *(cont'd)*

```
userNames.forEach(function(name) {

  const user = new iam.User(stack, name, {
    userName: name
  })

  user.addManagedPolicy({
    managedPolicyArn: stack.formatArn({
      service:      "iam",
      region:       "",
      resource:     "policy",
      resourceName: "BasicUserPolicy"
    })
  })
})
```

We're iterating over the array defined as a script-level constant

Constructs are built with a scope, a name, and a map of properties; the user variable is rebound on each iteration

Some attributes must be set via method call

The stack object provides several utility functions

CDK: Output

```
Resources:
  user16DC45E76:
    Type: AWS::IAM::User
    Properties:
      ManagedPolicyArns:
        - Fn::Join:
            - ""
            - - "arn:"
              - Ref: AWS::Partition
              - ":iam::"
              - Ref: AWS::AccountId
              - :policy/BasicUserPolicy
      UserName: user1
    Metadata:
      aws:cdk:path: UsersAndGroupsStack/user1/Resource
```

CDK generates logical IDs

This is just nasty

Provides some level of debuggability

(and so on)

But, wait, that’s not what CDK is good at!

Are You Deploying a Monolith?

Monolithic web-apps are usually deployed as a single stack

Micro-service architectures tend to deploy groups of related resources

- Compute
- Data storage
- Logging
- Permissions

Reusable components simplify your templates

Analogy: atoms versus molecules versus cells

What Goes Into Creating an SQS Queue?

What your developers care about:

- Queue name / URL

What you, the application architect, care about:

- Queue name
- Visibility timeout
- Redrive policy (includes creating a dead-letter queue)
- Retention period
- IAM policies (or policy snippets)
- Application tags

Approaches to Reusability

CloudFormation

- Nested stacks (soft limit of 200 stacks per account)
- Custom resources (moves your infrastructure definition into a Lambda)

Terraform

- Modules

CDK

- Higher-order constructs (including many provided by AWS)

Terraform Modules

- A mechanism for including one Terraform script inside another
  - Your main script is, in fact, a module, and can be included in another script
  - Modules are configured via variables, can provide information to caller via outputs
  - Convention: separate source files for variables, configuration, and outputs
- Modules can be defined within the project or sourced externally
  - Sources include Terraform Registry, Git repository, S3 bucket, and HTTP(S) server
- There are over 1,400 AWS modules available from Terraform Registry

Terraform Module Example: variables.tf

```
variable "queue_name" {
  description = "The name of the queue. Used as a prefix for related resource names."
  type = string
}

variable "retention_period" {
  description = "Time (in seconds) that messages will remain in queue before being purged"
  type = number
  default = 86400
}
```

(and so on)

Terraform Module Example: main.tf

```
provider "aws" {}

resource "aws_sqs_queue" "base_queue" {
  name                         = var.queue_name
  message_retention_seconds    = var.retention_period
  visibility_timeout_seconds   = var.visibility_timeout
  redrive_policy               = jsonencode({
                                     "deadLetterTargetArn" = aws_sqs_queue.deadletter_queue.arn,
                                     "maxReceiveCount" = var.retry_count
                                 })
}
```

(and so on)

Terraform Module Example: outputs.tf

```
output "base_queue_url" {
  value = aws_sqs_queue.base_queue.id
}

output "deadletter_queue_url" {
  value = aws_sqs_queue.deadletter_queue.id
}

output "consumer_policy_arn" {
  value = aws_iam_policy.consumer_policy.arn
}

output "producer_policy_arn" {
  value = aws_iam_policy.producer_policy.arn
}
```

Terraform Module Example: Invocation

```
provider "aws" {}

module "foo_queue" {
  source = "./modules/create_sqs"
  queue_name = "Foo"
}
…
resource "aws_iam_role_policy_attachment" "application_role_foo_producer" {
  role       = aws_iam_role.application_role.name
  policy_arn = module.foo_queue.producer_policy_arn
}
```

CDK Constructs

A class definition that subclasses `cdk.Construct`

- If your construct creates a single resource, subclass `cdk.Resource`

Can accept configuration parameters and expose attributes

- Attributes must be calculable at build-time, *do not* reference actual AWS resource properties

Can live in project lib directory, or as installable Node library

CDK Construct Example: Properties

```
export interface StandardQueueConfig {
    /**
     *  The name of the primary queue. This is used to construct the name
     *  of the dead-letter queue and consumer/producer policies.
     */
    readonly queueName: string
}
```

CDK Construct Example: Attributes

```
export interface IStandardQueue  extends cdk.IResource {

  /**
   * The primary queue
   * @attribute
   */
  readonly mainQueue: sqs.Queue;

  /**
   * The dead letter queue
   * @attribute
   */
  readonly deadLetterQueue: sqs.Queue;
```

(and so on)

CDK Construct Example: Construct

```
export class StandardQueue extends cdk.Construct {

  public readonly mainQueue: sqs.Queue;
  public readonly deadLetterQueue: sqs.Queue;
  public readonly consumerPolicy: iam.ManagedPolicy;
  public readonly producerPolicy: iam.ManagedPolicy;

  constructor(scope: cdk.Construct, id: string, props: StandardQueueConfig) {
    super(scope, id);

    this.mainQueue = new sqs.Queue(this, "Main", {
        queueName: props.queueName
    })
```

(and so on)

CDK Construct Example: Invocation

```
export class MultiQueueStack extends cdk.Stack {
  constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);
    const stack = this;

    const q1 = new StandardQueue(stack, "Foo", {
        queueName: "Foo"
    })
…
    const appRole = new iam.Role(sstackelf, "ApplicationRole", {
        roleName:        self.stackName + "-ApplicationRole",
        assumedBy:       new iam.ServicePrincipal('ec2.amazonaws.com'),
        managedPolicies: [ q1.producerPolicy, q2.producerPolicy, q3.producerPolicy ]
    })
  }
}
```

Closing Thoughts

Think About Your Module/Construct Design

Naming conventions

- `FooQueue`, `FooQueue-DLQ`, `SQS-FooQueue-ReaderPolicy`, `SQS-FooQueue-WriterPolicy`

Prefer convention over configuration

- The more consistency you have, the easier it is to maintain your code
- `Foo-DLQ` versus `Foo-DeadLetterQueue` … does the difference matter?
- Pick one style and move on!

IAM policy granularity

- If the queue is to be used by multiple applications, it deserves its own reader/writer policies
- If a single application uses multiple queues, it’s better to have a single application policy

Parameterize! (but not too much)

- Use default values to create human-readable constants
- Consistent parameter/variable names encourage automation
- But beware: templates with many parameters are hard to manage
  - And likely to hide security holes
- Consider externalizing configuration
  - CloudFormation *dynamic references* can access Parameter Store and Secrets Manager
  - Terraform also provides data sources that can retrieve these values (beware: stored in state!)
  - Some resources can retrieve configuration from Parameter Store/Secrets Manager directly

Leverage Runtime Information

Example: identifying public/private subnets for a VPC

- This can determined by looking at the route table associated with each subnet, to determine whether it has an Internet Gateway as the ultimate destination
- Could also tag subnets with “public” and “private”

Things like this are “easy” programmatically, difficult declaratively

- CloudFormation can use a custom resource (Lambda)
- Terraform provides some “data” objects
- CDK provides a “context” object

Or you could write a program to generate parameters/variables

Do It Yourself?

Anything that can produce JSON or YAML can create a CloudFormation template

There are several existing frameworks

- CFNDSL for Ruby
- Troposphere for Python

Opinion: most valuable when used by another script

Some Links

Example of creating users/groups with all three tools (plus CFNDSL)

https://github.com/chariotsolutions/aws-examples/tree/master/infrastructure-tools-comparison

Example of using modules/constructs to create SQS queues

https://github.com/chariotsolutions/aws-examples/tree/master/infrastructure-tools-comparison-2

CFRunner

A Python program that will create/update CloudFormations stacks, reading parameters from a file and appending outputs to that file.

https://github.com/kdgregory/aws-misc/blob/master/utils/cf-runner.py

Chariot Solutions

https://chariotsolutions.com/