AWS Cloud Development Kit

Infrastructure as Components

Infrastructure as Code (IaC) is hot, maar kan het nog *hotter*? In dit artikel maken we kennis met de AWS Cloud Development Kit (CDK) [1] waarmee we de AWS infrastructuur creëren voor een Gatling applicatie met ondersteuning voor realtime monitoring in Grafana. Het artikel bespreekt ervaringen met de CDK en biedt voldoende handvatten om te bepalen of de CDK toegevoegde waarde biedt voor jouw project.

Het inrichten en beheren van cloud infrastructuur heeft een enorme ontwikkeling doorgemaakt sinds de lancering van AWS in 2006. De eerste jaren gebeurde dit handmatig via de AWS Console of eigen tooling en scripts. Vanaf 2010 werd het mogelijk om infrastructuur als code (JSON of YAML) te schrijven met AWS CloudFormation. Rond 2015 ontstonden er projecten die CloudFormation templates genereren vanuit programmeertalen, bijvoorbeeld Troposphere (Python) en SparkleFormation (Ruby).

Tijdens AWS re:Invent 2018 introduceert AWS de Cloud Development Kit. De CDK maakt het schrijven van component-gebaseerde infrastructuur code in moderne programmeertalen mogelijk. Het biedt een *mental model* dat ons in staat stelt om (complexe) infrastructuur in begrijpelijke en overzichtelijke stukken te definiëren. Hiermee worden *software design principles* (abstractie en encapsulatie) toepasbaar op infrastructuur code en vervaagt de grens met applicatie code.

Wat is de AWS CDK?

De AWS CDK is een open source *polyglot* software development framework waarmee je AWS cloud infrastructuur in moderne programmeertalen schrijft (TypeScript, JavaScript, Java, C# of Python) en CloudFormation templates in JSON of YAML genereerd. Afbeelding 1 toont de architectuur van de CDK.

AWS CDK en CloudFormation vormen een sterke combinatie: de CDK verrijkt het schrijven van infrastructuur code met de voordelen van procedurele logica en objectgeoriënteerde technieken, terwijl CloudFormation de infrastructuur *state* beheert op basis van declaratieve CloudFormation templates die de gewenste eindtoestand van infrastructuur resources beschrijven.

De belangrijkste onderdelen van de AWS CDK zijn:

- CDK Core Framework
- CDK Construct Library
- CDK CLI

CDK Core Framework

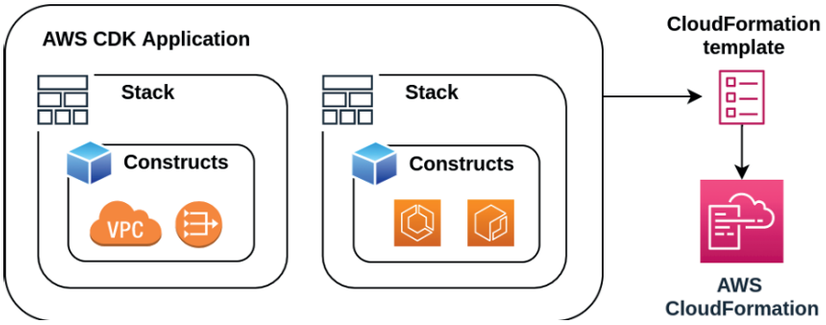
Het core framework bestaat uit de volgende onderdelen:

- Construct: basis bouwsteen waarin één of meerdere AWS resources zijn samengesteld tot een bruikbaar 'component'.
- Stack: *unit of deployment* in de CDK, opgebouwd uit een hiërarchische structuur van Constructs, de *Construct tree*.
- App: CDK applicatie opgebouwd uit één of meerdere Stacks waarin de infrastructuur en AWS cloud resources voor een systeem zijn gedefinieerd.

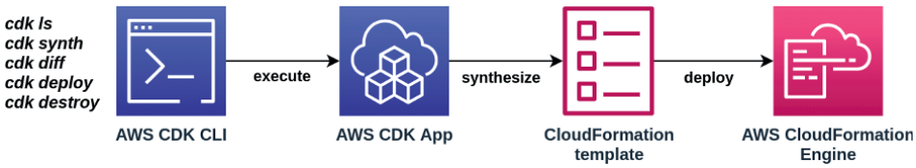
CDK Construct Library

De *AWS Construct Library* [2] geeft invulling aan de kern uitgangspunten van de CDK: abstractie en herbruikbaarheid. Het is een verzameling modules van *Constructs* die een API biedt om AWS resources te maken in CDK applicaties. Een module heeft betrekking op een specifieke AWS service, bijvoorbeeld aws-sqs (voor SQS) en aws-s3 (voor S3).

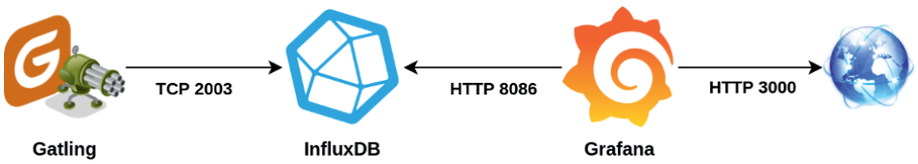
Christiaan Rudolfs is als Senior Java Developer van PanCompany werkzaam bij Malmberg. Hij heeft meer dan 15 jaar (internationale) ervaring als software ontwikkelaar en heeft een passie voor Java, Software Architectuur, Clean Code, (AWS) Cloud Development en ontdekt graag de toegevoegde waarde van nieuwe technologieën.



Afbeelding 1: AWS CDK architectuur.



Afbeelding 2: AWS CDK CLI workflow.



Afbeelding 3: Gatling realtime monitoring service architectuur.

HET INRICHTEN EN BEHEREN VAN CLOUD INFRASTRUCTUUR HEEFT EEN ENORME ONTWIKKELING DOORGEMAAKT SINDS DE LANCERING VAN AWS IN 2006

Een CDK applicatie is opgebouwd uit een hiërarchische structuur van Constructs: de *construct tree*. Er zijn drie Construct abstractieniveaus te onderscheiden:

1. **low-level constructs** (CFN Constructs): CloudFormation Native constructs zijn 1 op 1 representaties van een CloudFormation resource.
2. **high-level constructs**: Constructs met een hoger abstractieniveau, samengesteld uit een willekeurig aantal lower-level constructs die boilerplate code wegnemen en voorzien in sensible defaults.
3. **patterns**: Constructs met het hoogste abstractieniveau die oplossingen bieden voor alledaagse AWS infrastructuur ontwerpen.

CDK CLI

De CDK CLI is een Node.js applicatie ontwikkeld in TypeScript. De CLI voert de CDK applicatie uit en biedt commando's voor het tonen van alle stacks binnen de applicatie, voor het genereren van CloudFormation voor de stacks (*synthesize*) en voor het deployen en vernietigen van stacks. Het is ook mogelijk om de wijzigingen ten opzichte van de bestaande infrastructuur te zien via het *cdk diff* commando. Afbeelding 2 geeft weer hoe de AWS CDK CLI werkt.

Hoe gebruik je de AWS CDK?

Onze voorbeeld applicatie bestaat uit drie services (Afbeelding 3) die elk in een Docker container draaien:

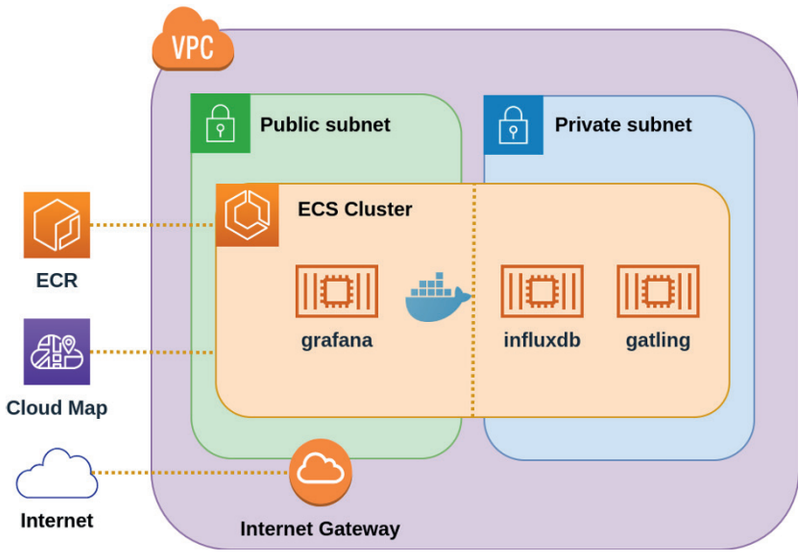
1. Gatling: voert load test scenario's uit
2. InfluxDB: persisteert load test data
3. Grafana: presenteert (real-time) load test resultaten

De services draaien in AWS Elastic Container Service (ECS) [3]. Een (vereenvoudigd) overzicht van de infrastructuur is weergegeven in Afbeelding 4.

Het ECS cluster vereist dat het in een Virtual Private Cloud (VPC) draait. De gatling en InfluxDB ECS services draaien beide in een private subnet en de grafana ECS service draait in een public subnet. Gatling en Grafana communiceren met InfluxDB via AWS Cloud Map service discovery service [4]. De Docker container images staan in AWS Elastic Container Registry (ECR) [5] van waaruit ECS deze kan deployen.

Show me some code!

Het is tijd om bovenstaande infrastructuur te bouwen met de AWS CDK voor Java. De volledige code voor dit project is beschikbaar in mijn github repository [6].



Afbeelding 4: Gatling realtime monitoring infrastructuur.

Onze CDK applicatie bestaat uit twee stacks, één waarin de VPC is gedefinieerd en één waarin het ECS cluster is gedefinieerd. Listing 1 toont de main class van de applicatie waarin de App en Stacks initialisatie plaatsvindt.

De code laat zien hoe de *construct tree* is opgebouwd. Een *Construct* wordt altijd gedefinieerd binnen de scope van een ander *Construct*, daarom wordt elk Construct object geïnitialiseerd met een *scope* en *id*. De App en Stack types zijn beide ook Constructs en we zien dat beide Stacks zijn gedefinieerd binnen de scope van de App. Daarmee vormt de App het *root* element van de *construct tree* met daaronder de Stacks die weer zijn opgebouwd uit Constructs.

Listing 2 geeft onze GatlingVpcStack weer, waarin een *high-level* Vpc construct wordt gebruikt. Dit is een mooi voorbeeld dat laat zien hoe je in een paar regels code complexe infrastructuur definieert. De *sensible defaults* in het Vpc construct zorgen ervoor dat er 1 public en 1 private subnet per availability zone gemaakt wordt en dat de opgegeven CIDR range evenredig wordt verdeeld over alle subnets. De output van dit stuk code resulteert in 23 verschillende AWS resources en genereert bijna 300 regels CloudFormation in YAML formaat!

Listing 3 toont een code fragment van de GatlingEcsStack met het ECS Cluster en één van onze drie ECS services. Je kunt ervoor kiezen om iedere ECS service als een logisch geheel in een apart construct te definiëren om zo de implementatie details van elkaar te scheiden.

Listing 4 toont een deel van de implementatie van één van de ECS service constructs, de GatlingRunnerFargateService. De code bestaat uit meerdere *high-level* constructs, waaronder een FargateTaskDefinition en FargateService. Er wordt *boilerplate* code weggenomen door het gebruik van *sensible defaults* in bijvoorbeeld de TaskDefinition, waarvoor standaard een *execution role* en *task role* gemaakt wordt op basis van het *least-privilege* principe.

Tot slot verdient het CDK concept *Assets* nog aandacht. Assets zijn lokale bestanden of Docker images die onderdeel uitmaken van de CDK applicatie. In ons voorbeeld zorgt *ContainerImage.fromAsset* ervoor dat tijdens het CDK deploy proces automatisch een Docker container image wordt gebouwd die

```
import software.amazon.awscdk.core.App;

// Instantiate the AWS CDK app
App app = new App();

// Instantiate GatlingVpcStack
IVpc vpc = new GatlingVpcStack(app, vpcStackName,
       stackProps, projectName).get();

// Instantiate ECS stack
GatlingEcsStack.builder().scope(app).id(ecsStackName)
       .stackProps(stackProps)
       .namespace(projectName)
       .vpc(vpc)
       .build();

// Produce the cloud assembly
app.synth();
```

Listing 1: initialisatie CDK App en Stacks.

```
import software.amazon.awscdk.core.Stack;
import software.amazon.awscdk.services.ec2.Vpc;

public class GatlingVpcStack extends Stack {
   private final Vpc vpc;

   public GatlingVpcStack(Construct scope, String id, StackProps
   stackProps, String namespace) {
       super(scope, id, stackProps);

       this.vpc = Vpc.Builder.create(this, "GatlingVpc")
               .cidr("10.12.0.0/16")
               .build();
   }
}
```

Listing 2: GatlingVpcStack.

```
Cluster ecsCluster = Cluster.Builder.create(this, "GatlingCluster")
       .clusterName(ecsClusterName)
       .vpc(vpc)
       .build();

GatlingRunnerFargateService.builder()
  .serviceProps(GatlingEcsServiceProps.builder()
                 .serviceName(serviceName)
                 .clusterNamespace(namespace)
                 .ecsCluster(ecsCluster)
                 .vpc(vpc)
                 .build())
  .build(this, "GatlingRunnerFargateService");

// similar code for GrafanaFargateService
// similar code for InfluxdbEc2Service
```

Listing 3: GatlingEcsStack.

vervolgens naar ECR wordt gepushed en aan de specifieke ECS Service wordt gekoppeld. De CDK bundelt op deze manier de applicatie en bijbehorende infrastructuur in één deployable cloud artifact. Deze stack bespaart ons het schrijven van 820 regels YAML!

Infrastructuur deployment

Nu is het tijd voor een deployment van onze CDK applicatie naar AWS. Installatie en prere-

quisites staan beschreven in de README van mijn repo [6].

Het commando `cdk ls` toont het stack overzicht: gatlingVpcStack en gatlingEcsStack. Voordat we de deployment naar AWS uitvoeren verifiëren we de wijzigingen via het commando `cdk diff gatlingEcsStack`. Dit geeft per stack de AWS resource en security wijzigingen weer. Hiermee wordt de kracht van het declaratieve state management van CloudFormation benut. Het commando `cdk deploy gatlingEcsStack` rolt beide CloudFormation stacks uit naar AWS met een duidelijke terugkoppeling over de voortgang. Verder bouwt het de drie Docker container images en pushed deze naar ECR. De CDK plaatst alle gegenereerde CloudFormation templates standaard in de cdk.out directory. De infrastructuur voor onze applicatie is nu volledig ingericht en de applicatie up and running. Afbeelding 5 toont het ECS Cluster in de AWS console.

Een punt van aandacht is het correct instellen van *IAM policies* (op basis van het *least privilege* principe) voor de AWS gebruiker waarmee de deployment uitgevoerd wordt. Dit is nu alleen mogelijk via *trial and error* en kost veel tijd. Het genereren van de benodigde set aan *IAM policies* zou een welkome aanvulling op de CDK zijn.

Toepasbaarheid

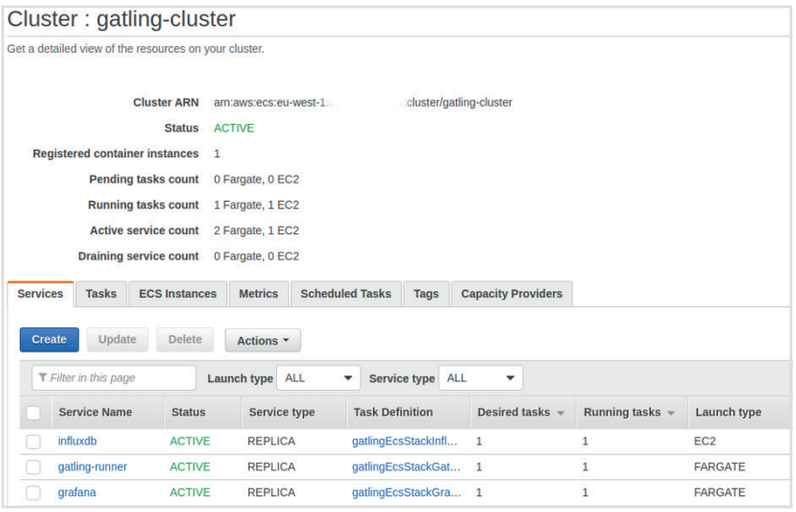
Hoe past de CDK binnen DevOps? Om de CDK effectief te gebruiken is er gedegen kennis nodig van de gebruikte programmeertaal (en concepten). Daarnaast is kennis van AWS resources en CloudFormation noodzakelijk om de impact van de gegenereerde CloudFormation op de infrastructuur te bepalen. Ik verwacht niet dat de gemiddelde Ops-er warm loopt om de CDK te gebruiken, aangezien in die rol software bouwen (en denken in abstracties) nu eenmaal niet een core competentie is. Aan de andere kant heeft de gemiddelde ontwikkelaar (Dev) in de praktijk vaak te weinig kennis van AWS en CloudFormation. Het gebruik van de CDK biedt daarom voorlopig vooral meerwaarde bij afgebakende (container gebaseerde of serverless) applicatiecomponenten waarbij er geen afhankelijkheid van Ops is en ontwikkelaars met voldoende AWS kennis beschikbaar zijn. In de praktijk zal CDK gebruik in projecten pas toenemen naarmate de 'kloof' tussen infrastructuur en applicatiesoftware kleiner wordt en teams met de juiste skills sets inzien dat een verenigd development proces leidt tot een efficiëntere software development lifecycle.

```
FargateTaskDefinition taskDef =
       FargateTaskDefinition.Builder.create(this, "GatlingFTD")
               .build();

String dockerFileDir = "../gatling-monitoring/gatling-runner";
taskDef.addContainer("gatling", ContainerDefinitionOptions.builder()
                       .image(ContainerImage.fromAsset(dockerFileDir))
                       .build());

FargateService.Builder.create(this, id)
       .serviceName(serviceName)
       .taskDefinition(taskDef)
       .cluster(ecsCluster)
       .build();
```

Listing 4: GatlingRunnerFargateService.

Conclusie

De AWS CDK biedt een toegankelijk framework voor ontwikkelaars om efficiënter en productiever infrastructuur code voor het AWS cloud platform te schrijven in bekende programmeertalen. De rijkelijk gevulde AWS Construct Library en actieve CDK community maken de CDK een waardevol product dat ik zeker weer zal gebruiken! ■



Afbeelding 5: ECS Cluster in AWS Console.

REFERENTIES

[1] AWS CDK documentatie: **https://docs.aws.amazon.com/cdk/latest/guide/home.html**

[2] AWS CDK Construct Library: **https://docs.aws.amazon.com/cdk/api/latest/docs/aws-construct-library.html**

[3] AWS ECS: **https://aws.amazon.com/ecs/**

[4] AWS Cloud Map: **https://aws.amazon.com/cloud-map/**

[5] AWS ECR: **https://aws.amazon.com/ecr/**

[6] Gatling AWS CDK code: **https://github.com/crudolfs/gatling-realtime-monitoring-aws-cdk**