Bericht

**Public Cloud Services HS23**

Dozent: Sebastian Graf

Studenten: Fabian Heuberger  
Etienne Frei  
Yannick Hohler

CI Pipeline on AWS

Ein Bild, das Rad, orange enthält.

Automatisch generierte Beschreibung

Inhaltsverzeichnis

[Use Case 3](#_Toc155614366)

[Architektur 4](#_Toc155614367)

[Umsetzung 5](#_Toc155614368)

[IaC 5](#_Toc155614369)

[AWS CodePipeline 5](#_Toc155614370)

[AWS S3 6](#_Toc155614371)

[AWS CodeBuild 6](#_Toc155614372)

[AWS SNS 7](#_Toc155614373)

[Amazon EventBridge 7](#_Toc155614374)

[AWS Lambda 7](#_Toc155614375)

[Erkenntnisse & Fazit 9](#_Toc155614376)

# Use Case

Wir haben uns entschieden, eine Continuous CI/CD-Pipeline für die Bereitstellung von Angular-Projekten zu implementieren. Die Bereitstellung unseres IaC-Codes wird in Github erfolgen. AWS CloudFormation wird die Infrastruktur direkt mit Änderungen am Quellcode über eine Codestar-Verbindung aktualisieren.

Eine solche Pipeline könnte über Github Actions umgesetzt werden, jedoch ist die Auslastung der von GitHub bereitgestellten Runner begrenzt und diese müssten bei einem grösseren Projekt mit vielen Änderungen durch einen eigenen Runner ersetzt werden.

Um die Pipeline an einem zentralen Ort zu haben, soll die ganze Pipeline direkt in AWS umgesetzt werden. Dies führt zu einer besseren Kontrolle des gesamten Prozesses und führt zu einer nahtlosen Verbindung zu anderen AWS Services. Das Monitoring und Logging wird dadurch auch erleichtert.

# Architektur

Ein Bild, das Text, Screenshot, Diagramm, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 1 Gesamte Architektur

* GitHub dient als Repository für die Codebase des Angular Projekts.
* CodePipeline orchestriert die Ausführung der Stages und dient als zentrales Element dieses Systems.
* CodeBuild welches von CodePipeline verwaltet wird erstellt die Artefakte der Applikation und speichert diese in einem S3 Bucket.
* Der SNS Dienst benachrichtigt per E-Mail über den Zustand der Pipeline
* EventBridge beobachtet die Ereignisse welche von der CodePipeline generiert werden und leitet diese weiter.
* Die Lambda Funktion erstellt eine Rückmeldung auf GitHub basierend auf Ergebnissen der Pipeline

# Umsetzung

In diesem Kapitel wird beschrieben wie die einzelne Services konfiguriert wurden und wie unser Service Stack erstellt wurde.

## IaC

Als Erstes haben wir versucht, Terraform als IaC-Tool zu verwenden. Es hat sich jedoch herausgestellt, dass es Funktionen gibt, die noch nicht unterstützt werden (Bug: [[Bug]: aws\_codebuild\_project queued\_timeout unsupported value when using lambda compute · Issue #34376 · hashicorp/terraform-provider-aws (github.com)](https://github.com/hashicorp/terraform-provider-aws/issues/34376)). Bspw. beim CodeBuild-Service haben wir AWS Lambda als "Compute Mode" gewählt. Dieser Modus unterstützt das Attribut "queued\_timeout" nicht. Beim Erstellen der Services wird ein Standardwert festgelegt, obwohl im Terraform-Template nichts definiert ist, was beim Bereitstellen der Services einen Fehler wirft.

Daher haben wir uns entschieden, AWS CloudFormation zu verwenden, um unsere Services zu erstellen. CloudFormation bietet folgende Vorteile:

* Es bietet eine tiefere Integration mit anderen AWS-Services und -Funktionen
* Die Syntax ist deklarativ
* CloudFormation bietet eine grafische Benutzeroberfläche namens AWS CloudFormation Designer

Um die Services zu konfigurieren haben wir diese zuerst über die AWS Konsole eingerichtet. Erst danach wurden diese in eine CloudFormation Skript umgewandelt. Dies erleichtert die Erstellung von IaC da über die Konsole klar definiert ist welche Eigenschaften definiert werden müssen und welche optional sind. Falls die Infrastruktur direkt mit IaC konfiguriert werden sollen, muss eine gewisse Erfahrung mit den Diensten vorhanden sein. Die Dokumentation für CloudFormation Skripte ist vollständig und verständlich. Als Anfänger sollte jedoch zuerst das Minimum definiert werden, bevor die Skripte erweitert werden um komplexes Debuggen zu verhindern.

Über einen Git Sync wird die Infrastruktur bei jedem Commit automatisch aktualisiert. Durch diese Feature können sehr einfach mehrere Stacks erstellt werden jeweils für Staging und dann auch für die Produktion zum Beispiel. Es könnnen auch Stacks in verschiedene Regions erstellt werden um die Availability der Services sicherzustellen.

## AWS CodePipeline

AWS CodePipeline ist ein Continuous-Delivery-Service, der zur Automatisierung von Software-Release-Prozessen eingesetzt wird. Unsere Pipeline besteht aus 3 Phasen:

* Source
* Build
* Deploy

**Source Phase**:

In dieser Phase greift der Pipeline-Prozess auf den neuesten Commit über eine Codestar-Verbindung, zieht die relevanten Dateien und speichert sie in die Artefakten-Bucket.

**Build Phase:**

Die Build-Phase holt sich die Artefakte aus dem Artefakten-Bucket und buildet das Projekt gemäss den in der Buildspec-Datei im Projekt-Repository spezifizierten Anweisungen. Dieser Prozess resultiert in der Erstellung eines Release-Dist-Verzeichnisses. Das generierte Release wird als ZIP-Datei im Artefakten-Bucket abgelegt.

**Deploy Phase:**

In der Deployment-Phase wird das erstellte Release-ZIP aus dem Artefakten-Bucket abgerufen. Dieses ZIP wird entpackt, und die enthaltenen Dateien werden als statische Website im Website-Bucket bereitgestellt. Die Webseite ist unter folgender URL erreichbar: <http://pipeline-stack-websitebucket-c2zd6dkpe7zm.s3-website-eu-west-1.amazonaws.com>

Durch diese Release Pipeline wird sichergestellt, dass jede Änderung im Quellcode erfolgreich gebaut und bereitgestellt wird, wodurch ein effizienter Development- und Delivery-Prozess gewährleistet wird.

## AWS S3

In unserem CodePipeline Projekt wurden folgende 3 S3 Buckets erstellt und verwendet:

* WebsiteBucket: ein öffentlicher Bucket für das deployen einer statischen Webseite: <http://pipeline-stack-websitebucket-c2zd6dkpe7zm.s3-website-eu-west-1.amazonaws.com>
* ArtifactBucket: der Bucket wird verwendet um alle Source und Build Artefakte zu speichern.
* Sources: in diesem Bucket ist der Quellcode der Lambda Funktion angelegt.

## AWS CodeBuild

AWS CodeBuild wurde für die Build Phase verwendet. Für das CodeBuild Project haben wir statt einer EC2 Instanz AWS Lambda für das Ausführen unsere Builds verwendet. AWS Lambda ermöglicht schnellere Builds aufgrund einer geringeren Startlatenz. AWS Lambda skaliert auch automatisch, sodass Builds nicht in der Warteschlange warten, bis sie ausgeführt werden.

Der Build wird durch die Buildspec Datei im Angular Projekt definiert:

Buildspec.yaml:

version: 0.2

env:

    variables:

        S3\_BUCKET: "{{S3\_BUCKET}}"

        CACHE\_CONTROL: "86400"

        BUILD\_FOLDER: "dist"

phases:

  install:

    runtime-versions:

        nodejs: 18

    commands:

        - echo Installing source NPM dependencies...

        - npm install

        - npm install -g @angular/cli

  build:

    commands:

        - echo Build started

        - ng build

artifacts:

    files:

        - '\*\*/\*'

    base-directory: 'dist\*'

    discard-paths: yes

### AWS SNS

Wir haben SNS verwendet, um Benachrichtigungen über den Status der Pipeline zu versenden. Um das zu erreichen haben wir eine Notifikationsregel für die Pipeline registriert und eine SNS Queue als Target gesetzt.

## Amazon EventBridge

Einen EventBridge ist ein zentraler Ereignisbus in der Ereignisse erfasst und weitergeleitet werden. Grundsätzlich besitzt jeder AWS Account eine Standard EventBridge. Auf dieser EventBridge können Regeln definiert werden die besagen welche Ereignisse an welche Services weitergeleitet werden. EventBridge ist serverlos und skalierbar.

Folgendes Ereignismuster löst die Regel aus: Ein Bild, das Text, Screenshot, Schrift, Reihe enthält.

Automatisch generierte Beschreibung

Abbildung Ereignismuster welches die definierte Regel auslöst

Sobald ein solches Ereignis aus der CodePipeline in EventBridge erscheint, wird die in der Regel verknüpfte Lambda Funktion damit aufgerufen. Wir haben uns entschieden die Zustände «suceeded» und «failed» der drei Phasen der CodePipeline für diese Case Study zu verarbeiten.

## AWS Lambda

Um direkt auf GitHub eine Rückmeldung zu generieren haben wir eine Lambda Funktion erstellt. Diese Funktion erstellt einen Kommentar unter dem Commit welche die Pipeline durchlaufen hat.   
Wie im Kapitel Amazon EventBridge beschrieben wird die Funktion durch ein Ereignis der CodePipeline ausgelöst welche von der Eventbridge weitergeleitet wird wie in Abbildung 2 dargestellt wird.

Ein Bild, das Text, Screenshot, Schrift, Reihe enthält.

Automatisch generierte Beschreibung

Abbildung Verknüpfung zwischen EventBridge und Lambda Funktion

Die Lambda Funktion haben wir in JavaScript geschrieben und ist als Zip-Datei in einem S3-Bucket abgelegt. Benötigte Abhängigkeiten mussten innerhalb des Ordners installiert werden.

Folgendes Ereignis wird von der Lambda Funktion empfangen:   
Ein Bild, das Text, Screenshot, Schrift, Dokument enthält.

Automatisch generierte Beschreibung

Dieses Ereignis enthält jedoch nicht den Hash des Commits. Dies kann mit der AWS-SDK Bibliothek umgesetzt werden indem eine CodePipeline-Instanz erstellt wird und mit den Parametern wie Name und Execution-Id weitere Details abgerufen werden können:

Ein Bild, das Text, Schrift, Reihe, Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung Extrahieren von Details einer CodePipeline Ausführung mit AWS-SDK

Mit dem Hash des Commits kann nun über die GitHub-API der Commit erfasst und der Kommentar erstellt werden.

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung

Abbildung Erstellung des Kommentar unter spezifischem Commit

Damit die Lambda Funktion die Berechtigungen besitzt Kommentare im jeweiligen Repository zu schreiben wurde ein Access Token eines der Accounts benötigt. Somit werden die Kommentare im Namen eines Contributors geschrieben.

Um die Services zu konfigurieren haben wir diese zuerst über die AWS Konsole eingerichtet. Erst danach wurden diese in eine CloudFormation Skript umgewandelt. Dies erleichtert die Erstellung von IaC da über die Konsole klar definiert ist welche Eigenschaften definiert werden müssen und welche optional sind. Falls die Infrastruktur direkt mit IaC konfiguriert werden sollen, muss eine gewisse Erfahrung mit den Diensten vorhanden sein. Die Dokumentation für CloudFormation Skripte ist vollständig und verständlich. Als Anfänger sollte jedoch zuerst das Minimum definiert werden, bevor die Skripte erweitert werden um komplexes Debuggen zu verhindern.

Lambda Funktionen können direkt über die AWS Konsole mit Testereignissen getestet werden, was die Erstellung erleichtert wie in Abbildung 6 zu sehen ist.

Ein Bild, das Text, Screenshot, Zahl, Schrift enthält.

Automatisch generierte Beschreibung

# Erkenntnisse & Fazit

Als Softwareentwickler war es sehr interessant zu sehen, dass man nicht nur mit einem Jenkins-Server oder über GitHub Actions einen effizienten Deployment-Prozess erstellen kann, sondern auch über AWS-Services, die sich sehr einfach skalieren lassen.

Alle Gruppenmitglieder hatten wenig Erfahrung mit IaC-Tools. Wir konnten jedoch mit Terraform und CloudFormation Erfahrungen sammeln. Auf der einen Seite war es schade, dass wir das Projekt nicht mit Terraform umsetzen konnten, weil der Terraform AWS Provider noch nicht alle Features unterstützt, die wir verwenden wollten (z. B. GitHub (Version 2)-Verbindungen, CodeBuild Lambda Computation Type, usw.).

Glücklicherweise konnten wir dadurch jedoch die Features kennenlernen, die AWS CloudFormation bietet. CloudFormation bietet ein sehr praktisches Feature, bei dem Stacks sehr einfach direkt über eine Verbindung zu einem GitHub-Repository erstellt und nach jedem Commit aktualisiert werden können. Die Stacks können auch Grafisch dargestellt werden.

Diese Erfahrungen bilden eine solide Grundlage für die zukünftige Arbeit mit Cloud-Infrastrukturen. Wir hoffen dass wir unsere Skills in diesem Bereich weiter schärfen können und freuen uns auf die Entwicklung und Implementierung von Cloud-Infrastrukturen in zukünftigen Projekten.