Skalierbare, kosteneffiziente und hochverfügbare vCard Generierung

Technischer Bericht für AWS Cloud Service-Architektur

Colin Herzog

08. Januar 2024

Contents

1 Use Case

1.1 Hintergrund

Codin GmbH ist beauftragt, ein System zur digitalen Erstellung von Visitenkarten (vCards) zu entwickeln. Die momentane manuelle Erstellung ist zeitaufwendig und anfällig für Fehler. Der Kunde möchte, dass beim Scannen von physischen Visitenkarten, die mit NFC- oder QR-Codes ausgestattet sind, eine Webseite geöffnet wird, die den Download oder Import der vCard ermöglicht.

1.2 Ziel

Das Ziel ist die Automatisierung des vCard-Erstellungsprozesses durch ein cloudbasiertes System, das dynamisch und effizient vCards generiert, sobald neue Benutzerdaten verfügbar sind. Dies soll eine skalierbare, kosteneffiziente und hochverfügbare Lösung darstellen. Das System sollte in der Lage sein, vCards automatisch zu erstellen, in der Cloud zu speichern und über öffentliche Links zugänglich zu machen, um manuelle Interventionen zu eliminieren.

2 Architektur

2.1 Übersicht

Die Architektur zielt auf eine robuste, skalierbare Lösung mit AWS für die Erstellung und Verwaltung von vCards.

2.2 Verwendete AWS Services

2.2.1 Amazon DynamoDB

Amazon DynamoDB dient als zentraler Speicherort für Kontaktdaten. Es ist eine NoSQL-DB, mit hoher Skalierbarkeit. Die Entscheidung für DynamoDB liegt darin, dass per Stream Events ausegelesen werden können, tiefe Laufzeit- und Nutzungskosten bestehen. Das Pay-Per-Request-Preismodell ermöglicht eine effiziente Kostenkontrolle.

2.2.2 AWS Lambda

AWS Lambda wird verwendet, um die Logik zur Verarbeitung der Kontaktdaten mit dem API Gateway und zur Generierung der vCards zu implementieren. Die Lambda-Funktionen ermöglichen eine serverlose Architektur, wodurch die Kosten für Serverwartung und -betrieb entfallen. Diese Funktionen werden durch Änderungen in der DynamoDB-Tabelle ausgelöst und verarbeiten die Daten, um die vCards zu erstellen und in S3 zu speichern. Lambda bietet automatische Skalierung, um eine hohe Verfügbarkeit und Leistung zu gewährleisten.

2.2.3 Amazon S3

Amazon S3 wird für die Speicherung der generierten vCards und den Terraform State verwendet. Es bietet eine dauerhafte, skalierbare und sichere Speicherlösung. Durch die Verwendung von S3 sind die vCards dauerhaft gespeichert und können für das gesamte Internet bereitgestelt werden.

2.2.4 Amazon SES

Amazon SES wird genutzt, um Benutzer via E-Mail über die Erstellung der vCard informieren. SES ist ein flexibler und kosteneffizienter E-Mail-Dienst. Die Nutzung des SES kann ausgeweitet werden, sodass die Druckersoftware für die Karte die E-Mail automatisiert verarbeiten kann und einen automatischen Druck der Visitenkarte mit direkter Beschreibung des NFC-Links ermöglicht.

2.2.5 Amazon CloudWatch

Amazon CloudWatch wird für das Monitoring und Logging der Anwendung verwendet. CloudWatch sammelt und überwacht Log-Dateien, setzt Alarme und reagiert automatisch auf Änderungen in den AWS-Ressourcen. Dies ist entscheidend, um die Systemgesundheit zu überwachen und schnell auf mögliche Probleme reagieren zu können.

2.2.6 Amazon API Gateway

Amazon API Gateway spielt eine zentrale Rolle in der Architektur, indem es als Schnittstelle für den Benutzerzugriff auf die Funktionalitäten der vCard-Erstellung dient. Es ermöglicht das Definieren von HTTP-Endpunkten, an die Benutzeranfragen gesendet werden. Diese Anfragen werden dann an AWS Lambda-Funktionen zur weiteren Verarbeitung weitergeleitet.

2.3 Architektonische Prinzipien

Die Architektur folgt den bewährten Prinzipien des Cloud-Designs, um Skalierbarkeit, Kosteneffizienz und Hochverfügbarkeit sicherzustellen. Kernaspekte sind:

1. Serverlose Architektur: Durch die Verwendung von AWS Lambda und DynamoDB werden Serververwaltung und damit verbundene Betriebskosten bestmöglicht eliminiert.

2. Nutzung von Managed Services: Dienste wie S3, SES und CloudWatch reduzieren den Wartungsaufwand und steigern die Zuverlässigkeit.

3. Automatisierung und Skalierung: Die Architektur ist so konzipiert, dass sie sich nahtlos an wechselnde Lastanforderungen anpasst, ohne manuelle Eingriffe zu erfordern.

2.4 Alternativen und Optimierungen

Im Entwurfsprozess wurden Alternativen wie der Einsatz von Amazon EC2-Instanzen für die vCard-Generierung oder Amazon RDS für die Datenspeicherung in Betracht gezogen. Diese Optionen wurden jedoch zugunsten einer serverlosen Architektur verworfen, um die Effizienz und Wartbarkeit zu maximieren. Laufende Optimierungen, wie die Anpassung von Ressourcen und Feinabstimmung der Konfigurationen, tragen weiter zur Steigerung der Effizienz bei.

3 Umsetzung

3.1 IAC mit Terraform

Für die Implementierung der Services wurde IaC-Ansatz mit Terraform verwendet. Dies ermöglicht eine reproduzierbare und versionierbare Infrastrukturkonfiguration. Die Verwendung von Terraform unterstützt eine konsistente und fehlerfreie (wenn fehlerfrei Konfiguriert) Einrichtung der benötigten Ressourcen und erleichtert zukünftige Änderungen und Skalierungen.

3.1.1 Terraform Konfigurationen

Die Terraform-Konfigurationen umfassen die Definition von Ressourcen wie DynamoDB-Tabellen, Lambda-Funktionen, S3-Buckets, SES-Identitäten und IAM Rollen. Durch die Modularisierung der Terraform-Skripte wird die Wartbarkeit und Lesbarkeit verbessert. Jeder AWS-Dienst wird in einem eigenen Terraform-Modul konfiguriert, was die Wiederverwendbarkeit und das Verständnis der Konfigurationen erleichtert. Die Module sind (mit Ausnahme des Lambdas) immer gleich strukturiert. Auszug aus der Ordnerstruktur:

```
/
├── README.md
└── terraform
    ├── README.md
    ├── main.tf
    ├── outputs.tf
    ├── variables.tf
    └── modules
        ├── cloudwatch
        │   ├── README.md
        │   ├── main.tf
        │   ├── output.tf
        │   ├── variables.tf
        │   ├── lambda_function_payload.zip
        │   └── lambda_function_payload
        │       ├── api_handler.py
        │       └── dynamodb_trigger.py
        ├── api_gateway
        │   ├── README.md
        │   ├── main.tf
        │   ├── output.tf
        │   └── variables.tf
        ├── cloudwatch
        │   └── ...
        ├── dynamodb
        │   └── ...
        ├── iam
        │   └── ...
        ├── s3
        │   └── ...
        └── ses
            └── ...
```

Jeglicher AWS Service wurde mit Terraform definiert, mit ausnahme des Terraform State Buckets. Dieser muss händisch zuerst erstellt werden, bevor er als Umgebungsvariable im GitHub Repository eingetragen werden kann.

3.2 CI/CD Pipeline für Terraform Infrastruktur

3.2.1 Übersicht

Die Terraform Infrastructure Change Management Pipeline wird durch Änderungen im Main-Branch oder Pull Requests aktiviert.

3.2.2 Umgebungsvariablen

Wichtige Umgebungsvariablen wie `AWS ACCESS KEY ID`, `AWS SECRET ACCESS KEY`, `REGION` und `BUCKET TF STATE` werden aus den GitHub Secrets gezogen, um eine sichere Interaktion mit und vollständige Konfiguration der AWS-Dienste zu gewährleisten.

3.2.3 Jobs und Schritte

Die Pipeline besteht aus mehreren Schritten:

- **Checkout**: Der Code wird auf den Runner ausgecheckt.
- **Terraform Setup**: Terraform wird mit der spezifizierten Version auf dem Runner eingerichtet.
- **Initialisierung**: terraform init wird ausgeführt, wobei der Terraform-Zustand im spezifizierten S3-Bucket (muss bereits erstellt sein und als GitHub Variable `BUCKET TF STATE` gespeichert sein) gespeichert wird.
- **Formatierung**: Überprüfung der Terraform-Dateiformatierung mit terraform fmt.
- **Validierung**: Durchführung der terraform validate zur Überprüfung der Konfiguration.
- **Plan**: Erstellung eines Terraform-Plans, speziell bei Pull Requests.
- **Apply**: Bei Pushes im Main-Branch wird terraform apply automatisch ausgeführt, um Änderungen anzuwenden.

3.2.4 Terraform State Management

Der Terraform State wird zentral in einem S3-Bucket verwaltet. Diese Konfiguration ermöglicht es Terraform, den Zustand der Infrastruktur zwischen verschiedenen Ausführungen konsistent und sicher zu speichern. Die Schlüsselkonfigurationen dafür sind:

- `bucket`: Definiert den Namen des S3-Buckets, der für die Speicherung des Terraform-Zustands verwendet wird.

- `key`: Spezifiziert den Dateinamen im S3-Bucket, unter dem der Zustand gespeichert wird. Dieser S3-Bucket kann nicht mit einer Variable definiert werden (gemäss Spezifikation).
- `region`: Gibt die AWS-Region an, in der sich der S3-Bucket befindet.
- `encrypt`: Aktiviert die Verschlüsselung des gespeicherten Zustands, um die Sicherheit zu erhöhen.
- `dynamodb table`: Verweist auf die DynamoDB-Tabelle, die für das Locking des Zustands verwendet wird, um parallele Ausführungen zu verhindern.

Diese Konfiguration stellt sicher, dass der Terraform-Zustand zwischen verschiedenen Ausführungen und eventuellen Teammitgliedern synchronisiert ist.

3.3 Lambda-Funktionsimplementierung in Python

Die Lambda-Funktionen, die für die vCard-Erstellung verantwortlich sind, wurden in Python entwickelt.

3.3.1 Verarbeitung

Die Lambda-Funktionen reagieren auf Ereignisse aus dem API Gateway und des DynamoDB Streams, um bei Änderungen in den Benutzerdaten eine neue vCard zu erstellen. Der API Handler schreibt zuerst die Benutzerdaten in die DB und der trigger löst die zweite Funktion aus. Diese generiert die vCard basierend auf den Benutzerdaten, lädt sie in den entsprechenden S3-Bucket hoch und sendet eine E-Mail Benachrichtigung über SES.

3.3.2 Error Handling und Logging

Fehlerbehandlung und Logging sind entscheidende Bestandteile der Lambda-Funktionen. Fehler werden abgefangen und protokolliert, um die Nachvollziehbarkeit und schnelle Behebung von Problemen zu ermöglichen. Logs werden in CloudWatch gespeichert, was eine einfache Überwachung und Alarmierung ermöglicht. Ausserdem hilft das Error Logging bei der (weiter-) Entwicklung des Services.

4 Erkenntnisse

4.1 Herausforderungen und Lösungen

Während der Implementierung steiss ich auf diverse Herausforderungen, darunter die effiziente Verarbeitung von DynamoDB Streams durch Lambda und die Sicherstellung der Skalierbarkeit und Hochverfügbarkeit des Systems. Diese Herausforderungen wurden durch Recherche, Testen und Optimierung der Konfigurationen bewältigt.

4.2 Kosteneffizienz

Die gewählte Architektur und Implementierung hat sich als kosteneffizient erwiesen. Die Nutzung serverloser Dienste und das Pay-As-You-Go-Modell von AWS haben dazu beigetragen, die Betriebskosten niedrig zu halten, ohne die Leistung oder Verfügbarkeit zu beeinträchtigen. Die Erstellung von vCards wird im Verhältnis zum Scan einer Karte nicht oft durchgeführt. Die durchschnittlichen Daten zeigen pro Änderung einer vCard über 2'000 Scans. Bei einer durchschnittlichen Nutzung der Karte von 1-Mal pro Tag pro Karte ist dies eine sehr kleine Zahl. Diese Voraussetzungen bedingen tiefe Kosten wenn die App nicht läuft, was mit der Serverless Archtiektur sehr gut gedeckt wird.

4.3 Skalierbarkeit und Hochverfügbarkeit

Die Verwendung von AWS Lambda und DynamoDB bietet eine hohe Skalierbarkeit, da beide Dienste automatisch skaliert werden. Die Hochverfügbarkeit wird durch die robuste Infrastruktur von AWS sichergestellt.

4.4 Architektur Terraform

Die Nutzung von Terraform zur Gestaltung der Architektur hat sich als äusserst vorteilhaft erwiesen. Die Möglichkeit, die gesamte Cloud-Infrastruktur als Code zu definieren, hat zu einer erheblichen Steigerung der Effizienz, Transparenz und Wiederverwendbarkeit geführt. Einige Schlüsselerkenntnisse aus der Arbeit mit Terraform sind:

Modularität: Die Aufteilung der Infrastruktur in logische Module hat die Wartung und Skalierung vereinfacht. Jedes Modul kann unabhängig aktualisiert, wiederverwendet und getestet werden, was zu einer klaren Trennung der Verantwortlichkeiten führt.

Versionierung und Zusammenarbeit: Durch die Verwendung von Git für Terraform-Konfigurationen wurde eine effiziente Versionskontrolle ermöglicht. Dies unterstützt die

Zusammenarbeit im Team, indem Änderungen nachvollziehbar und konfliktfrei zusammengeführt werden können.

Change Management und Planung: Terraform's Planungs- und Apply-Schritte bieten eine klare Übersicht über Änderungen, bevor diese angewendet werden. Dies hat geholfen, unerwartete Auswirkungen zu vermeiden und die Sicherheit bei der Bereitstellung zu erhöhen. Die Planung und Bereitstellung kann mittels klar definierter CI / CD Pipeline kontrolliert werden.

4.5 CI / CD

Die Implementierung eines Continuous Integration und Continuous Deployment (CI/CD) Prozesses für die Infrastruktur und Anwendung war ein Schlüsselelement für die Effizienz und Zuverlässigkeit des Projekts.

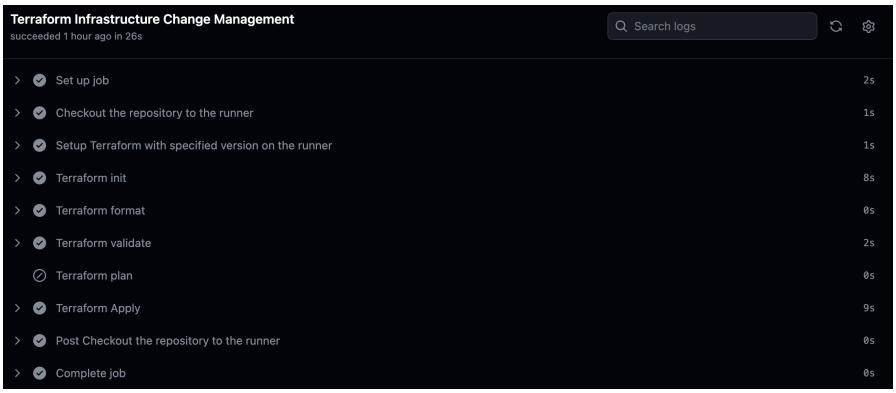

Figure 4.1: Schritte der GitHub Actions mit dazugehöriger Laufzeit

Bedeutende Punkte hierbei waren:

1. **Automatisierte Tests und Bereitstellung:** Die Einrichtung von Pipelines, die automatische Tests der Infrastruktur und des Codes durchführen und dann Änderungen in einer kontrollierten und vorhersehbaren Weise anwenden, hat die Qualität und Zuverlässigkeit des Deployments erheblich verbessert.

2. **Feedback und schnelle Iterationen:** Die CI/CD-Pipeline ermöglichte schnelle Iterationen und sofortiges Feedback zu Änderungen, was zur Beschleunigung der Entwicklung und zur frühzeitigen Identifikation von Problemen beigetragen hat.

3. **Infrastruktur- und Anwendungs-Deployment:** Die Integration von Terraform in den CI/CD-Prozess hat nicht nur die Bereitstellung der Anwendung, sondern auch der zugrundeliegenden Infrastruktur automatisiert. Dies sorgt für Konsistenz zwischen der Entwicklungs-, Test- und Produktionsumgebung.