



Praxisarbeit

Observability und Monitoring von AWS-Prozessen

Prüfer(in):

Prof. Dr. Christian Soltenborn

Verfasser(in):

Thomas Benjamin Hopf

101485

Heinrichstraße 23, 40764 Langenfeld

Wirtschaftsinformatik

Cyber Security

Eingereicht am:

26.08.2024

Sperrvermerk

Diese Arbeit enthält vertrauliche Informationen über die Firma Bayer AG. Die Weitergabe des Inhalts dieser Arbeit (auch in Auszügen) ist untersagt. Es dürfen keinerlei Kopien oder Abschriften - auch nicht in digitaler Form - angefertigt werden. Auch darf diese Arbeit nicht veröffentlicht werden und ist ausschließlich den Prüfern, Mitarbeitern der Verwaltung und Mitgliedern des Prüfungsausschusses sowie auf Nachfrage einer Evaluierungskommission zugänglich zu machen. Personen, die Einsicht in diese Arbeit erhalten, verpflichten sich, über die Inhalte dieser Arbeit und all ihren Anhängen keine Informationen, die die Firma Bayer AG betreffen, gegenüber Dritten preiszugeben. Ausnahmen bedürfen der schriftlichen Genehmigung der Firma Bayer AG und des Verfassers.

Die Arbeit oder Teile davon dürfen von der FHDW einer Plagiatsprüfung durch einen Plagiatsoftware-Anbieter unterzogen werden. Der Sperrvermerk ist somit im Fall einer Plagiatsprüfung nicht wirksam.

Inhaltsverzeichnis

Sperrvermerk	II
Abbildungsverzeichnis	IV
Tabellenverzeichnis	V
Listingverzeichnis	VI
1 Einleitung und Motivation	1
1.1 Zielsetzung	1
1.2 Unternehmen	1
1.3 Relevanz	2
1.4 Einschränkung	2
2 Grundlagen	3
2.1 Konzepte	3
2.1.1 Application Programming Interface (API)	3
2.1.2 Effizienzfaktoren	3
2.2 Werkzeuge	4
2.2.1 AWS	4
2.2.2 DynamoDB	4
2.2.3 GraphQL	4
2.2.4 AppSync	4
2.2.5 Terraform	4
2.2.6 Lambda	5
2.2.7 DataDog	5
2.2.8 CloudWatch	5
3 Anforderungen	6
3.1 Datengrundlage	6
3.2 Die API	7
3.2.1 Abfrage “GetUser”	7
3.2.2 Abfrage “GetUsers”	7
3.2.3 Abfrage “AddUser”	7
3.3 Fehlerbehandlung in der API	8

3.4	Datenverteilung	8
3.4.1	Theorie der Integration	9
3.4.2	Lambda-Funktion	9
3.5	Dashboard	9
3.5.1	Effizienz	10
3.5.2	Effektivität	10
3.6	CloudWatch	10
4	Umsetzung	11
4.1	Datenbank-Werkzeuge	11
4.1.1	DynamoDB	11
4.2	API	11
4.2.1	GraphQL	11
4.2.2	API-Methoden	12
4.2.3	GetUser	15
4.2.4	Lambda	17
4.3	AWS-Infrastruktur	22
4.3.1	Terraform	22
4.3.2	AWS-Datadog-Integration	26
4.4	Monitoring	27
4.4.1	DataDog	27
4.4.2	CloudWatch	29
5	Fazit	32
5.1	Bedeutung für das Unternehmen	32
5.2	Ausblick	32
5.3	Bewertung des Erreichens der Zielstellung	32
	Anhang	33
	Quellenverzeichnis	34
	Ehrenwörtliche Erklärung	35

Abbildungsverzeichnis

Abbildung 1: Beispiel einer JSON-Antwort	8
Abbildung 2: Datentypen	11
Abbildung 3: Status	12
Abbildung 4: Query und Mutation	12
Abbildung 5: Import der Bibliothek	13
Abbildung 6: Methode Request	13
Abbildung 7: Methode Response	14
Abbildung 8: Alternative Methode Response	14
Abbildung 9: Methoden GetUser	15
Abbildung 10: Methoden GetUsers	16
Abbildung 11: Methoden AddUser	17
Abbildung 12: GetLogger-Methode	18
Abbildung 13: GetSecret-Methode	18
Abbildung 14: Query-Methode	18
Abbildung 15: Definition der Query	19
Abbildung 16: Formierung des Requests	19
Abbildung 17: Durchführung der Anfrage	19
Abbildung 18: Fehlerhafte Query	20
Abbildung 19: Generieren des API-Keys	20
Abbildung 20: Ausführen der Query	20
Abbildung 21: Generieren des Loggers	21
Abbildung 22: Rückgabe des Felds “errors”	21
Abbildung 23: Korrekte Query	21
Abbildung 24: Rückgabe des korrekten Resultats	22
Abbildung 25: Definition der Tabelle	23
Abbildung 26: Definition der Berechtigungen	24
Abbildung 27: Definition der Funktions-Resolver	25
Abbildung 28: Aktivierung des Cloud-Watch-Loggings	26
Abbildung 29: Zeitstrahl; Einstellungen	28
Abbildung 30: Zeitstrahl	28
Abbildung 31: Tortendiagramm Filter	29
Abbildung 32: Tortendiagramm	29
Abbildung 33: LogStream	30

Abbildung 34: Filter für den LogStream	30
Abbildung 35: LogStream Beispiel	31

Tabellenverzeichnis

Tabelle 1: Aufbau der Tabelle	7
---	---

Listingverzeichnis

1 Einleitung und Motivation

Um den Nutzern Zugriff auf verschiedene Datensätze mit einer API zu ermöglichen, verwendet Bayer CropScience unter anderem „Amazon Web Services“ (AWS). Die Hintergrundprozesse in der Datenverarbeitung und Bereitstellung sind jedoch nicht einsichtig; wird eine Information aus einem Datensatz angefordert, lässt sich nur einsehen, ob der Prozess korrekt abgelaufen ist oder nicht, wodurch bei Auftreten von Fehlern die Behebung dieser nur schwer möglich ist. Zudem ist bisher keine Oberfläche implementiert, mit der die Antwortgeschwindigkeit (im Folgenden *Response time*) im zeitlichen Kontext überwacht werden kann. Diese beiden Faktoren wirken sich negativ auf die Effizienz und Effektivität der API aus, weshalb die in der Zielsetzung beschriebene Lösung implementiert werden soll.

1.1 Zielsetzung

Im Rahmen dieser Praxisarbeit soll die Effizienz und Effektivität der API gesteigert werden, indem die oben beschriebenen Maßnahmen entwickelt und eingebaut werden. Einsicht in den Prozess soll durch die Applikation „CloudWatch“ möglich gemacht werden, wobei Fehler genau dokumentiert werden; zusätzlich soll die Auftrittshäufigkeit von Fehlern in einem Dashboard dargestellt werden. Auch die Effizienz soll durch die Applikation „WatchDog“ überwacht werden. Hierzu sollen relevante Daten zu den jeweiligen Anfragen gesammelt und dargestellt werden, sodass auch Trends erkennbar sind.

1.2 Unternehmen

Bayer, gegründet 1863 als „Friedr. Bayer et comp.“, mit Sitz in Leverkusen ist das weltweit umsatzstärkste Unternehmen im Bereich CropScience. Aufgeteilt ist das Unternehmen in drei Divisionen: Pharmaceuticals, also der Herstellung verschreibungspflichtiger Arzneimittel, Consumer Health, die Herstellung sogenannter „Over-the-Counter Medikamente“, für welche keine ärztliche Verschreibung nötig ist, und CropScience, der Pflanzenschutzdivision. Zusätzlich zu den drei Divisionen existieren die „Enabling Functions“. Diese unterstützen das Tagesgeschäft, sind aber keiner der Divisionen direkt untergeordnet, sondern stehen neben den Divisionen und unterstützen diese.

Diese Praxisarbeit behandelt ein Projekt, welches primär für Crop-Science Nutzer relevant ist, jedoch im Auftrag einer Abteilung der „Enabling Functions“ durchgeführt wird.

1.3 Relevanz

Steigerung von Effizienz und Effektivität im Hintergrund der API führt zu einer verbesserten Nutzerfreundlichkeit. Durch Benachrichtigungen können Fehler schneller behoben werden, wodurch die Uptime erhöht wird, und durch Monitoring der Response time kann bei negativen Trends dieser früher gehandelt werden. Dadurch soll die API insgesamt am Ende, gestützt von den zu implementierenden Anwendungen, zuverlässiger laufen.

1.4 Einschränkung

Aus zeitlichen Gründen wird im Rahmen dieser Praxisarbeit die Implementierung nur auf einem Test-System durchgeführt, und alle beschriebenen Komponenten sind so nur für Testzwecke aufgebaut. Die Daten haben dementsprechend nicht immer semantische Relevanz, solange eine solche keinen Einfluss auf die Feststellung der Funktionalität der Infrastruktur haben. Die Anwendungen, mit denen im Rahmen dieser Praxisarbeit gearbeitet wird, sind Teil der Standard-Suite der Abteilung, die den Auftrag für diese Arbeit erteilt hat. Es hat keine eigene Auswahl bei diesen Anwendungen stattgefunden, weshalb die Auswahl der Anwendungen auch kein Thema in dieser Arbeit ist.

2 Grundlagen

Im Folgenden sollen Begriffe und Konzepte erklärt werden, die für das Verständnis dieser Arbeit vorausgesetzt sind. Dazu gehören sowohl die Metriken, mit denen die Zielerreichung festgestellt werden soll, als auch das theoretische Wissen, auf denen das Projekt aufbaut.

2.1 Konzepte

2.1.1 Application Programming Interface (API)

APIs sind Schnittstellen, die Entwicklern von Anwendungen die Möglichkeit bieten, auf Daten einer der Anwendung fernen Datenquelle sowohl lesend als auch schreibend zuzugreifen. Masse, Mark (2011) Bei geheimen Daten kann es dazu kommen, dass ein Entwickler nur mittels eines sogenannten „Disposable Tokens“ Zugriff auf die Daten hinter der Schnittstelle erlangen kann. Dabei wird ein Token generiert, welches Clients authentifiziert. Mit dieser Authentifizierung können auch verschiedene Berechtigungen wie „Read-Access“ (Nur lesen) oder „Read-Write-Access“ (Lesen und Schreiben) mit einem Nutzer assoziiert werden. Dieses System trägt dazu bei, die Datenintegrität der Daten zu gewährleisten.

2.1.2 Effizienzfaktoren

Kosten werden in AWS auf Basis der aufgewendeten Ressourcen berechnet. Kavis, M. J. (2014) Deshalb ist der Faktor Kosten für die Evaluierung potenzieller Effizienz relevant. Einerseits sollen Abfragen eine akzeptable und stabile Response time ermöglichen, gleichzeitig sollen die dafür nötigen Ressourcen möglichst klein gehalten werden. Es ist dementsprechend nicht zielführend, für eine höhere Effizienz die Rechenleistung und somit die Kosten zu erhöhen; viel eher soll der Verarbeitungsprozess bei Bedarf vereinfacht und optimiert werden, sodass die Response time ohne Erhöhung der Rechenkapazität verbessert wird. Services, Amazon Web (2024d)

2.2 Werkzeuge

2.2.1 AWS

AWS ist eine Cloud-Computing Plattform des Unternehmens Amazon, welches unter anderem Funktionen in der Datenverarbeitung, Datenspeicherung und Monitoring anbietet. Unter anderem Teil des AWS-Portfolios sind die Anwendungen „CloudWatch“, „AppSync“ und „DynamoDB“, welche im Laufe dieses Projekts Verwendung finden werden. Keesler, J. Baron H. Bond S. Davis B. Rydahl T. (2017)

2.2.2 DynamoDB

DynamoDB ist eine vollständig verwaltete Schlüssel-Wert- und Dokumentendatenbank, die bei jeder Größenordnung eine Responsetime von wenigen Millisekunden garantiert

2.2.3 GraphQL

GraphQL ist eine Query-Sprache mit denen sich Abfragen für eine API konfigurieren lassen. Durch die Begrenzung der Ausgabe auf explizit in der Query geforderte Elemente werden Ressourcen gespart. E. Porcello, A. Banks (2018a)

2.2.4 AppSync

AppSync bietet Werkzeuge zur Erstellung flexibler Schnittstellen, durch die ein sicherer Zugriff und Manipulation von Daten einer oder mehrerer Datensätze möglich ist.

2.2.5 Terraform

Terraform ist eine Konfigurationssprache, mit der eine Infrastruktur aufgesetzt wird, die Multi-Cloud-Bereitstellung automatisiert. Brikman, Y. (2019)

2.2.6 Lambda

Lambda ist eine Applikation des AWS-Systems, mit welcher automatische Datenverarbeitung auf Basis von selbst geschriebenen Methoden in fast jeder Art der Anwendung oder jedem Backend-Service ermöglicht. Services, Amazon Web (2024e) 3

2.2.7 DataDog

DataDog ist, ähnlich wie CloudWatch, eine Applikation mit Fokus auf der Überwachung von Ressourcen und Applikationen, jedoch mit Fokus auf Performance. Unter anderem durch Graphen lassen sich mit DataDog Trends in der Performance erkennen, wodurch präventive Maßnahmen getroffen werden können. Dixon, J. (2018)

2.2.8 CloudWatch

CloudWatch ist ein ist eine Applikation des AWS-Portfolios, mit welcher Ressourcen und Applikationen, die in der AWS-Umgebung laufen, in Echtzeit überwacht werden können. Unter anderem werden Fehler dokumentiert, Hinweise und andere Logs gespeichert, und wenn gewünscht Alarme bei bestimmten Fehlern ausgelöst. Services, Amazon Web (2024a) Services, Amazon Web (2024c) . Services, Amazon Web (2024b)

3 Anforderungen

Im Folgenden sollen die durchzuführenden Maßnahmen definiert werden, mit denen die Zielstellung erfüllt werden soll. An erster Stelle der Infrastruktur stehen die Daten selbst. Diese sind in einer mit DynamoDB aufgesetzten Datenbank gespeichert. Im Gegensatz zu unter anderem relationalen Datenbanken, in welchen neben dem Schlüssel auch jede Spalte genau definiert ist, bietet DynamoDB sogenannte „noSQL“-Datenbanken an, die neben einem Schlüsselement keine weiteren Vorgaben an den Spaltenaufbau machen. Daran muss die Datenverarbeitung mit entsprechenden Vorgaben und Kontrollen angepasst werden, um Laufzeitfehler durch Datentypkonversionen oder fehlende Daten zu vermeiden. Auf diese Daten greift eine API zu, welche mit AppSync entwickelt wurde. Sie bietet Nutzern die Möglichkeit, Daten je nach Bedarf zu lesen, bearbeiten oder hinzuzufügen. Im Hintergrund der API werden sowohl Terraform als auch GraphQL angewendet. Mit Terraform werden die API und ihre Anfragen-, mit GraphQL das erwartete Datenschema der Datenbank definiert, auf dessen Basis die API-Anfrage verarbeitet wird. Daten zu dieser API werden in zwei Richtungen weitergeleitet. Zum einen werden jegliche in der API auftretende Fehler an CloudWatch gesendet, wo diese dann verarbeitet werden, und zuständige Entwickler alarmiert. Zum anderen werden Daten über die zeitliche Effizienz an DataDog gesendet, wo sie in einem Dashboard verarbeitet werden, über welches die Entwickler die Performance der API prüfen, und eventuell bei schlechter Performance handeln können. Sollte die Vermutung bestehen, dass Teile der API fehleranfälliger sind, kann auf dem Dashboard in DataDog ebenfalls eine Übersicht über auftretende Fehler mit ihrer Häufigkeit und Gefahr aufgesetzt werden.

3.1 Datengrundlage

Die Basis der API bildet die DynamoDB-Tabelle „user-db“ (in Zukunft „Tabelle“), mit der E-Mail-Adresse des Nutzers als Schlüsselement. Zwar ist dies nicht von DynamoDB vorgeschrieben, jedoch sind in dieser Implementation die Spalten ebenfalls (auf semantischer Ebene) festgelegt, um die Komplexität der Datenverarbeitung kontrollieren zu können. Dazu sind die Informationen, die zu einem Nutzer abgespeichert werden in diesem Kontext konstant, weshalb flexible Spalten keinen Vorteil bieten.

Die Spalten sind folgendermaßen aufgeteilt:

Tabelle 1: Aufbau der Tabelle

Spalte		Beschreibung
E-Mail	E-Mail-Adresse des Nutzers	
Name	Name des Nutzers	
Status	Aktivitäts-Status (Aktiv/Inaktiv)	

Quelle: Eigene Darstellung

3.2 Die API

3.2.1 Abfrage “GetUser”

Als Parameter für diese Abfrage erwartet die API einen String, welcher die E-Mail-Adresse beinhalten soll zu der die Nutzerdaten angefordert werden. Zudem wird nach dem GraphQL-Standard erwartet, dass alle Spalten angegeben werden, deren Ausgabe vom Nutzer erwünscht sind.

3.2.2 Abfrage “GetUsers”

Hier erwartet die API nur die auszugebenen Spalten als Parameter. Ausgegeben werden dann die angeforderten Spalten aller Nutzer, die in der Tabelle hinterlegt sind.

3.2.3 Abfrage “AddUser”

Die API erwartet hier für jede Spalte der Tabelle jeweils einen String. Dazu erwartet die API eine Angabe zu den Spalten, die nach der Erstellung des Nutzers ausgegeben werden sollen. Mit diesen Informationen wird dann ein Eintrag in der Tabelle erstellt, und die gewünschten Spalten des erstellten Nutzers ausgegeben.

3.3 Fehlerbehandlung in der API

Eine Eigenschaft von GraphQL ist, dass Fehler nicht zum Abbruch der Abfrage führen. Tritt ein Fehler auf, wird dieser genau wie ein korrektes Ergebnis als JSON ausgegeben. E. Porcello, A. Banks (2018b) Der Aufbau der GraphQL-Antwort sieht folgendermaßen aus (die Beschreibungen der Felder werden durch Hovern angezeigt):

Abbildung 1: Beispiel einer JSON-Antwort

```
{  
  "data": [],  
  "errors": []  
}
```

Eigene Darstellung

Tritt ein Fehler auf, so ist das Feld „data“ leer, und die aufgetretenen Fehler werden in dem Feld „errors“ gelistet. Tritt keiner auf, befindet sich die Antwort auf die Anfrage in dem Feld „data“. Foundation, GraphQL (2024) Eine Nebeneigenschaft dieser Art der Fehlerbehandlung durch GraphQL ist, dass semantische Fehler in der Eingabe nicht zwingendermaßen abgefangen werden müssen, da die Abfrage trotzdem ausgeführt werden kann. Bei Bedarf kann eine Hilfestellung für Nutzer beim Auftreten häufiger Fehler eingebaut werden. Diese Fehlerbehandlung bringt jedoch eine Hürde in der Fehlererkennung in Datadog, da alle Antworten, ob fehlerhaft oder nicht, den gleichen Fehlercode ausgeben.

3.4 Datenverteilung

Die Infrastruktur für die AWS-Interne Weiterleitung von Daten besteht schon. Um Daten der API in ein Datadog-Dashboard zu integrieren, muss jedoch die Infrastruktur selbst aufgebaut werden.

3.4.1 Theorie der Integration

Zuerst muss die Verbindung zwischen dem AWS-Account, in dessen Namen auch die API und Tabelle erstellt wurden, und Datadog geknüpft werden. Dafür bietet Datadog eine Oberfläche. Um die Verbindung herzustellen, werden zwei Tokens generiert: ein “API-Key” und ein “Application-Key”. Mit diesen beiden Tokens kann dann in einer bestimmten Region die Schnittstelle seitens Datadog geöffnet werden. Die Schnittstelle seitens AWS ist über das Tool „CloudFormation“ möglich. Dort muss ein Stack erstellt werden, welcher ebenfalls mit dem generierten Application-Key versehen wird, wodurch dann die beiden Schnittstellen miteinander verbunden sind. Ein Datenfluss existiert jedoch trotzdem nicht, da es in der Testumgebung keine Endnutzer gibt; dieser muss manuell in einer Programmiersprache über eine Lambda-Funktion implementiert werden. In diesem Fall wird die Sprache Python verwendet.

3.4.2 Lambda-Funktion

Für Testzwecke werden im Rahmen dieser Praxisarbeit zwei Lambda-Funktionen erstellt. In der einen Funktion wird eine korrekte, in der anderen eine inkorrekte Abfrage simuliert. Dabei werden Daten zu der Ausgabe und der Antwort-Zeit weitergegeben. Diese können dann in DataDog als Dashboard umgesetzt werden.

3.5 Dashboard

Da die Alarmierung bei Fehlern durch CloudWatch abgedeckt ist, muss das DataDog-Dashboard diese nicht abdecken. Um Effizienz und Effektivität kontrollieren zu können, sind die Minimalanforderungen ein Graph für die Response-Time und ein Graph über die Fehlerhäufigkeit. Im Rahmen dieser Praxisarbeit werden jeweils eine KPI für Effizienz und Effektivität abgedeckt, während weitere bei Bedarf in der Zukunft hinzugefügt werden können. dabei wird als Hauptkriterium die Rückmeldung der Endnutzer von Relevanz sein.

3.5.1 Effizienz

Anhand einer Timeline soll das Dashboard dem Endnutzer ermöglichen, die KPI “Response time” zu überprüfen. **DatadogAWS2024**. Das soll anhand eines Zeitgraphen erreicht werden, auf dem die Antwortzeit der Anfragen gemessen werden. Damit Trends leichter erkennbar sind, soll ebenfalls eine Trendlinie über den Graphen gelegt werden.

3.5.2 Effektivität

Das Dashboard soll dem Endnutzer ermöglichen, das Verhältnis zwischen erfolgreichen und fehlerhaften Anfragen einzusehen, damit zu jedem Zeitpunkt ein Eindruck über die Funktionalität der API möglich ist.

3.6 CloudWatch

Nachdem die API in AWS integriert wurde, löst CloudWatch bei Fehlern selbstständig Alarme aus, weshalb keine weitere Infrastruktur aufgebaut werden muss. Die Logs werden danach sofort gesammelt, und lösen entsprechende Alarme ausgelöst.

4 Umsetzung

4.1 Datenbank-Werkzeuge

4.1.1 DynamoDB

4.2 API

4.2.1 GraphQL

Der GraphQL-Code erfüllt zwei verschiedene Aufgaben **GraphQLAWSAppSync2024**: Er definiert das Schema der Datenbank: zwar sind in DynamoDB keine Spalten vorgegeben, eine spaltenunabhängige Codierung ist im Rahmen dieses Projekts aber weder semantisch vorteilhaft noch im Rahmen der Praxisarbeit zeitlich realisierbar. Deshalb wird die Tabelle mit den jeweiligen Datentypen vorgegeben:

Abbildung 2: Datentypen

```
type User {  
  email: String!  
  name: String!  
  status: Status  
}
```

Eigene Darstellung

Bezogen auf den Nutzer werden drei Attribute gespeichert: Die email als String, der Name als String und der Status. Dieser ist entweder “ACTIVE” oder “INACTIVE”. Um das zu erreichen muss in GraphQL ein sogenannter “enum” erstellt werden. Enums sind Strukturen, die eine Reihe verschiedener Optionen enthalten. Es werden nur Werte akzeptiert, die so auch in der Struktur gespeichert sind (In diesem Fall ACTIVE und INACTIVE). <https://graphql.com/learn/enums/>

Abbildung 3: Status

```
enum Status {
  ACTIVE
  INACTIVE
}
```

Eigene Darstellung

Zusätzlich dazu werden in GraphQL die Funktionen der API definiert. Den Anforderungen (s.o.) entsprechend muss es drei Funktionen, eine Mutation und zwei Abfragen geben. Die Abfrage “GetUsers” benötigt keine Eingabe, da ohne Selektion alle in der Datenbank enthaltenen Einträge ausgegeben werden. Die Abfrage “GetUser” hingegen benötigt die E-Mail als Filterparameter, und die Mutation AddUser braucht alle 3 Informationen über den Nutzer. Das muss in der Definition berücksichtigt werden:

Abbildung 4: Query und Mutation

```
type Query {
  getUsers: [User]
  getUser(email: String!): User
}

type Mutation {
  addUser(email: String!, name: String!,
    status: Status): User
}
```

Eigene Darstellung

4.2.2 API-Methoden

Für jede API-Methode existiert eine JavaScript-Datei. Die Methoden unterscheiden sich aufgrund ihrer unterschiedlichen Anforderungen.

Bibliothek Um Daten aus der Tabelle mit JavaScript zu editieren, wird eine Bibliothek verwendet, in der verschiedene Standard-Operationen enthalten sind, die auf einer DynamoDB-Tabelle ausgeführt werden können. Auf Basis dieser Bibliothek sind dann alle Methoden aufgebaut. Die Bibliothek wird in allen Dateien unter dem gleichen Alias

importiert, weshalb der Import kein Teil der Code-Snippets für die Methoden ist, und stattdessen im Folgenden beispielhaft isoliert dargestellt wird. Der Name, mit der die Bibliothek referenziert wird, ist “ddb”:

Abbildung 5: Import der Bibliothek

```
import * as ddb from  
  ↪ '@aws-appsync/utils/dynamodb'
```

Eigene Darstellung

Aufbau der Dateien Da in allen JavaScript-Dateien jeweils eine API-Funktion definiert wird, ist der grundlegende Aufbau identisch. Die Dateien unterscheiden sich lediglich durch den Inhalt der Methode selbst. Auf den oben beschriebenen Import der Bibliothek folgend wird die Funktion “request” definiert. Die Funktion dient der Behandlung der Anfrage, das Resultat der Methode wird als “result” in der Kontext-Variable gespeichert. Die Kontext-Variable wird als Input-Parameter in die Methode gegeben und beinhaltet alle relevanten Informationen rund um die Anfrage. Das Resultat hat keinen vorgegebenen Datentyp:

Abbildung 6: Methode Request

```
export function request(ctx) {  
  //handle the request  
  //return the result  
}
```

Eigene Darstellung

Nachdem die Abfrage durch die Request-Methode behandelt wurde, muss auch die Ausgabe behandelt werden. Das geschieht über die Funktion “response”. Sollte keine weitere Mutation des Resultats nötig sein, sieht diese Methode folgendermaßen aus:

Abbildung 7: Methode Response

```
export const response = (ctx) =>  
  ↪ ctx.result;
```

Eigene Darstellung

Der Eingabe-Parameter für die Methode ist ctx, also die Kontext-Variable. Ausgegeben wird ctx.result, die Variable in der das Resultat gespeichert wird. Ist ctx.result leer, so wird “undefined” zurückgegeben. Die Methode ist also äquivalent zu der Folgenden:

Abbildung 8: Alternative Methode Response

```
export const response = (ctx) => {  
  if (ctx !== null) return ctx.result;  
  else return undefined;  
};
```

Eigene Darstellung

4.2.3 GetUser

Aus den Anforderungen an diese Methode geht hervor, dass zu den Eingabe-Parametern die email gehören muss, damit nach dieser gefiltert werden kann. Zusätzlich muss noch die Möglichkeit geboten werden, bis zu drei weitere Variablen mitzugeben. Diese stellen dann die Felder dar, die von der API zurückgegeben werden. Für die Ausgabe muss definiert werden, welche Operation ausgeführt wird. Die Antwort ist dann die Rückgabe der Abfrage mit dem Filterparameter:

Abbildung 9: Methoden GetUser

```
export function request(ctx) {  
  return ddb.get({ key: { email:  
    ↪ ctx.args.email } });  
}  
  
export const response = (ctx) =>  
  ↪ ctx.result;
```

Eigene Darstellung

Für die Abfrage wird die Methode `ddb.get()` verwendet, welche die entsprechend passenden Einträge für einen gegebenen Schlüsselparameter liefert, sofern diese vorhanden sind.

GetUsers Aus den Anforderungen an diese Methode geht hervor, dass als Eingabe-Parameter nur die Möglichkeit bestehen muss, die auszugebenden Felder auszuwählen. Für die Ausgabe muss definiert werden, welche Operation ausgeführt wird,

Die Antwort ist dann die Rückgabe der Abfrage:

Abbildung 10: Methoden GetUsers

```
export function request(ctx) {  
  return { operation: 'Scan' };  
}  
  
export const response = (ctx) =>  
  ↪ ctx.result.items;
```

Eigene Darstellung

Aufgrund eines Fehlers mit der Methode `ddb.scan()` wird hier die Bibliothek nicht eingesetzt. Stattdessen wird die Scan-Operation manuell aufgerufen, und das Ergebnis dieser zurückgegeben. Die Antwort wird dann aus allen Items der `ctx-result`-Variable gebildet.

AddUser Aus den Anforderungen an diese Methode geht hervor, dass als Eingabe-Parameter zumindest die email, also der Schlüssel der Tabelle, gegeben sein muss. Dazu muss die Möglichkeit bestehen, Inhalt für alle anderen Spalten mitzugeben, und Spalten zur Ausgabe auszuwählen. Die Ausgabe bei dieser Mutation ist das durch die Methode selbst hinzugefügte Element:

Abbildung 11: Methoden AddUser

```

export function request(ctx) {
  const { email, ...values } = ctx.args
  return {
    operation: 'PutItem',
    key:
      ↪ util.dynamodb.toMapValues({email}),
    attributeValues:
      ↪ util.dynamodb.toMapValues(values),
  };
}
export const response = (ctx) =>
  ↪ ctx.result;

```

Eigene Darstellung

Zuerst werden hier die Argumente extrahiert, da diese gesondert genutzt werden. Dafür wird `ctx.args`, die Variable in der alle Argumente gespeichert sind, auf zwei Variablen verteilt. Das erste Argument (in diesem Fall die `email` als Schlüssel) wird auf die Variable `email` übertragen, alle weiteren Argumente auf die Liste `values`. Danach wird die Operation definiert, hier “PutItem”, also das Hinzufügen eines Elements, woraufhin die Schlüsselspalte und die anderen Spalten auf die vorher gespeicherten Variablen gesetzt werden. Die Response-Methode ist identisch zu der, die in `GetUser` verwendet wurde.

4.2.4 Lambda

Im Rahmen der Lambda-Funktionen werden drei verschiedene Dateien verwendet. In der Datei “`shared.py`” sind die Methoden hinterlegt, die für alle Funktionen relevant sind. Die Dateien “`success.py`” und “`error.py`” beinhalten jeweils eine Lambda-Funktion, wobei erstere eine korrekte, und zweitere eine fehlerhafte Anfrage simuliert.

Geteilte Methoden Für die Funktionen werden drei verschiedene unterstützende Methoden benötigt (in diesem speziellen Anwendungsfall, diese Zahl ist nicht unbedingt auf

andere Projekte übertragbar). Mit einer Methode wird die Instanz eines sogenannten “logger”s verteilt. Ein logger ist dafür zuständig, Nachrichten über die Ausführung eines Programms auszugeben:

Abbildung 12: GetLogger-Methode

```
def get_logger(level: int = logging.INFO):  
    logger = logging.getLogger()  
    logger.setLevel(level)  
    return logger
```

Eigene Darstellung

Eine weitere Methode ist dafür zuständig, das API-Secret zu generieren, mit welchem sich die Methoden authentifizieren:

Abbildung 13: GetSecret-Methode

```
def get_secret(secret_id: str) -> str:  
    response =  
        ↪ sm_client.get_secret_value(SecretId=secret_id)  
    return response["SecretString"]
```

Eigene Darstellung

Die letzte Methode ist dafür zuständig die Query auszuführen, die in den beiden anderen Dateien definiert wird. Als Eingabe-Parameter werden die URL der angesprochenen Schnittstelle, der API-Key und die query erwartet:

Abbildung 14: Query-Methode

```
def query(url: str, key: str, query: str):
```

Eigene Darstellung

Die Bearbeitung ist in mehrere Schritte gegliedert:

1. Definition der query:

Abbildung 15: Definition der Query

```
data = json.dumps({"query":  
    ↪ query}).encode()
```

Eigene Darstellung

2. Formierung des requests, indem das Request selbst mit Methode und URL und der Header mit Content-Type und API-Key definiert werden:

Abbildung 16: Formierung des Requests

```
req = request.Request(url, method="POST")  
req.add_header("x-api-key", key)  
req.add_header("Content-Type",  
    ↪ "application/graphql")
```

Eigene Darstellung

3. Durchführung der Anfrage und Rückgabe der Antwort:

Abbildung 17: Durchführung der Anfrage

```
response = request.urlopen(req, data=data)  
return json.loads(response.read())
```

Eigene Darstellung

Simulation der fehlerhaften Abfrage Basis der fehlerhaften Anfrage ist eine fehlerhafte Query. Diese sieht folgendermaßen aus:

Abbildung 18: Fehlerhafte Query

```
QUERY = """
query MyQuery {
  getUsers {
    email
    name
    stat
  }
}
"""
```

Eigene Darstellung

Das Feld “stat” existiert nicht, weshalb hier in der Ausführung ein Fehler ausgeworfen wird.

Zusätzlich beinhaltet die Datei eine Methode, mit der die Query behandelt wird. Diese teilt sich folgendermaßen auf:

1. Generieren des API-Keys mithilfe der get secret-Methode aus “shared.py”:

Abbildung 19: Generieren des API-Keys

```
key =
↪ get_secret(os.environ["API_KEY_SECRET_ID"])
```

Eigene Darstellung

2. Ausführen der Query über die Methode aus “shared.py”:

Abbildung 20: Ausführen der Query

```
result = query(os.environ["API_URL"], key,
↪ QUERY)
```

Eigene Darstellung

3. Generieren des loggers mithilfe der Methode aus “shared.py” und Dokumentation des Fehlers:

Abbildung 21: Generieren des Loggers

```
get_logger().error(result)
```

Eigene Darstellung

4. Rückgabe des Felds “errors” aus dem Resultat:

Abbildung 22: Rückgabe des Felds “errors”

```
return result["errors"]
```

Eigene Darstellung

Simulation der erfolgreichen Abfrage Basis der erfolgreichen Abfrage ist eine korrekte Query. Diese sieht folgendermaßen aus:

Abbildung 23: Korrekte Query

```
QUERY = """
query MyQuery {
  getUsers {
    email
    name
    status
  }
}
"""
```

Eigene Darstellung

Genau wie die Simulation der fehlerhaften Abfrage wird die Query auch hier in einer separaten Methode behandelt. Diese ist bis auf die Rückgabe identisch. Zurückgegeben wird nicht der Fehler sondern das korrekte Resultat:

Abbildung 24: Rückgabe des korrekten Resultats

```
return result["data"]["getUsers"]
```

Eigene Darstellung

4.3 AWS-Infrastruktur

4.3.1 Terraform

Die Basis der in Terraform definierten Infrastruktur ist die Verbindung mit der Datenbank und die Definition der zu verwendenden Rollen für die jeweiligen Abfragen.

Definition der Tabelle Zuerst wird dafür eine Resource erstellt, in welcher die von Terraform angeforderten Informationen über die Datenbank angegeben werden **TerraformBestPractices2024**. Dazu gehören der Name der Datenbank, der Zahlungsmodus und der Datenbankschlüssel. Beim Zahlungsmodus unterscheidet AWS zwischen “provisioned” und “pay per request”. Bei unvorhersehbaren Anfragefrequenzen empfiehlt AWS “pay per request”, weshalb dieses auch hier verwendet wird: Da die Umsetzung auf dem Testsystem stattfindet, ist nicht von regelmäßiger Nutzung auszugehen. Der Name der Datenbank ist “users-db” und der Datenbankschlüssel ist die E-Mail. Mit diesen Informationen kann nun die Ressource erstellt werden:

Abbildung 25: Definition der Tabelle

```
resource "aws_dynamodb_table" "user" {  
    name           = "user-db"  
    billing_mode   = "PAY_PER_REQUEST"  
    hash_key       = "email"  
  
    attribute {  
        name = "email"  
        type = "S"  
    }  
}
```

Eigene Darstellung

Definition der Berechtigungen Da die API zwei verschiedene Methoden-Typen anbietet, die jeweils verschiedene Zugriffslevel benötigen, müssen auch zwei verschiedene Zugriffspolitiken aufgesetzt werden. Für Queries muss die API Daten aus der Tabelle lesen können, und für die Mutation die Tabelle editieren. Die beiden Zugriffsberechtigungen sind vom Aufbau her gleich, weshalb im Folgenden nur zur Veranschaulichung der Schreib-Zugriff beschrieben wird. Die Basis bildet hier ein “IAM policy document”. Dieses JSON-Dokument enthält Informationen, ohne die der Zugriff nicht gewährt werden kann, und bildet somit die Quelle des Zugriffs. Zusätzlich zu dieser Quelle müssen manuell die Aktionen definiert werden, die mit dieser Rolle verbunden sein sollen, und es muss eine Identifikationsnummer für die Rolle vergeben werden. Diese Informationen werden als Datenobjekt abgespeichert, und können dann von der API zur Authorisierung verwendet werden:

Abbildung 26: Definition der Berechtigungen

```
data "aws_iam_policy_document"
  ↪ "user_table_write" {
    source_policy_documents =
      ↪ [data.aws_iam_policy_document.user_table_read.json]

    statement {
      sid = "WriteItems"
      actions = [
        "dynamodb:PutItem",
        "dynamodb:UpdateItem",
        "dynamodb>DeleteItem",
      ]
      resources =
        ↪ [aws_dynamodb_table.user.arn]
    }
  }
}
```

Eigene Darstellung

Definition der Funktions-Resolver Für jede Funktion der API muss in Terraform ein sogenannter Resolver angelegt werden. In einem Resolver sind Informationen über die Methode hinterlegt, die ausgeführt werden soll. Enthalten muss ein Resolver den Typ der Funktion, die Identifikationsnummer der API, die Datengrundlage, den Namen der Funktion, die Art der Funktion, und die Datei, in der die Methode beschrieben wird. Im Rahmen dieser Praxisarbeit sind die Methoden in Javascript definiert¹. Zuletzt muss noch die Runtime der Sprache, in der die Methoden verfasst sind, definiert werden, damit die Methoden ausgeführt werden können. Der Resolver wird dann als Resource abgespeichert. Beispielhaft ist hier die Methode AddUser dargestellt:

¹Die Sprache hat keinen Einfluss auf die Funktionsweise, die Entscheidung beruht ausschließlich auf der Präferenz der Entwickler

Abbildung 27: Definition der Funktions-Resolver

```
resource "aws_appsync_resolver"
↪  "add_user" {
    type          = "Mutation"
    api_id        =
    ↪  aws_appsync_graphql_api.user.id
    data_source =
    ↪  aws_appsync_datasource.user_table.name
    field         = "addUser"
    kind          = "UNIT"
    code          =
    ↪  file("${path.module}/templates/addUser.js")

    runtime {
        name          = "APPSYNC_JS"
        runtime_version = "1.0.0"
    }
}
```

Eigene Darstellung

Aktivierung des Cloud-Watch-Loggings Um das Sammeln von Daten in CloudWatch zu aktivieren, muss eine Methode in Terraform auf die entsprechende Policy referenzieren

Abbildung 28: Aktivierung des Cloud-Watch-Loggings

```

resource "aws_iam_role_policy_attachment"
  ↪ "logging" {
    policy_arn =
      ↪ "arn:aws:iam::aws:policy/service-role/AWSAppSyncPushToCloudWatchLogs"
    role       =
      ↪ aws_iam_role.logging.name
  }

```

Eigene Darstellung

4.3.2 AWS-Datadog-Integration

Die praktische Umsetzung der Integration beschränkt sich auf die Eingabe von Daten in den Benutzeroberflächen von DataDog und AWS.

DataDog Zuerst muss in DataDog ein neuer AWS-Account als Integration hinzugefügt werden. Dabei werden dem Nutzer vier Anpassungsoptionen gegeben:

1. AWS-Region

In diesem Fall, da die Auftraggebende Abteilung auf diesem Server beheimatet ist, ist die Region “eu-central-1”

2. API Key

Lässt der Nutzer hier das Feld unverändert, generiert DataDog automatisch einen API-Key. Für dieses Projekt wurde bereits im Vorfeld ein API-Key angelegt, der deshalb hier ausgewählt wird.

3. AWS Logs senden

Damit die im vorigen Kapitel aufgesetzten Lambda-Funktionen einen Nutzen haben, sollte diese Option aktiviert sein.

4. Cloud Security Verwaltung anschalten

Im Rahmen dieser Praxisarbeit ist Sicherheit nicht ausschlaggebend, da keine schützenswerten Daten behandelt werden. Die Kosten für Sicherheit werden dementsprechend gespart und diese Option bleibt deaktiviert.

Nachdem die Felder ausgefüllt wurden, kann das Template erstellt werden. Damit ist der in DataDog zu absolvierende Teil abgeschlossen.

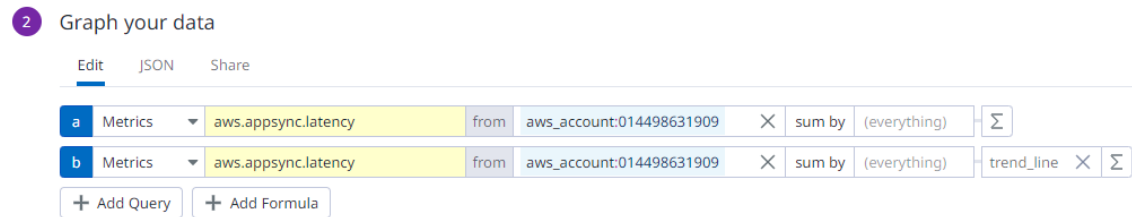
AWS Mit der Fertigstellung des Templates in DataDog wird der Nutzer automatisch in die Applikation “CloudFormation” weitergeleitet. Dort kann dann mit dem API-Key aus DataDog und einem App-Key, welcher ebenfalls in DataDog generiert wird, ein Stack erstellt werden. Damit ist die Verbindung abgeschlossen und Daten können zwischen AWS und DataDog fließen.

4.4 Monitoring

4.4.1 DataDog

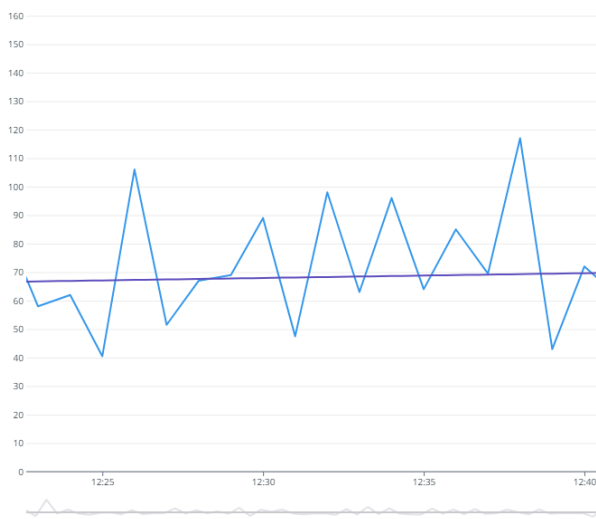
Aus den Anforderungen gehen drei verschiedene zu implementierende Module hervor. Das erste ist ein Zeitstrahl, mit dem die Response time im zeitlichen Kontext dargestellt werden kann. Das andere ist ein Diagramm, durch welches eine Relation zwischen erfolgreichen und nicht erfolgreichen Abfragen hergestellt werden kann. Hier bietet sich ein Tortendiagramm an, ein Balkendiagramm, Strahlendiagramm oder sogar ein Zeitstrahl für Tendenzen wären auch denkbar. Die Auswahl basiert hier nicht darauf, welches Diagramm am besten für den Nutzungsfall geeignet ist. Diese Entscheidung ist auch von Nutzerpräferenz abhängig, und kann aufgrund des frühen Entwicklungsstadiums noch nicht miteinbezogen werden kann.

Zeitstrahl Für den Zeitstrahl wird die Metrik “aws.appsync.latency” verwendet. Ein Strahl stellt dabei die exakte Response time zu einem bestimmten Zeitpunkt dar, während der andere die zugehörige Trendlinie bildet, die das Visualisieren von Trends vereinfachen soll:

Abbildung 29: Zeitstrahl; Einstellungen

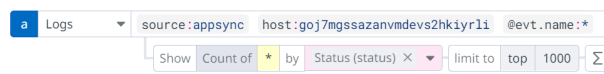
Eigene Darstellung

Hier ein beispielhafter Ausschnitt des Graphen:

Abbildung 30: Zeitstrahl

Eigene Darstellung

Tortendiagramm Zur Umsetzung des Tortendiagramms müssen die Filter so gewählt werden, dass mit einer einzigen OR-Abfrage informative logs von fehlerhaften und erfolgreichen Abfragen (im Folgenden Antworten) unterschieden werden können. Unter anderem unterscheiden sich die beiden JSONs der Antwort-Logs von den informativen logs darin, dass in Informativen Logs kein Event existiert. Die variable “evt.name” existiert nur in Antwort-Logs. Der Filter wird also folgendermaßen aufgebaut:

Abbildung 31: Tortendiagramm Filter

Eigene Darstellung

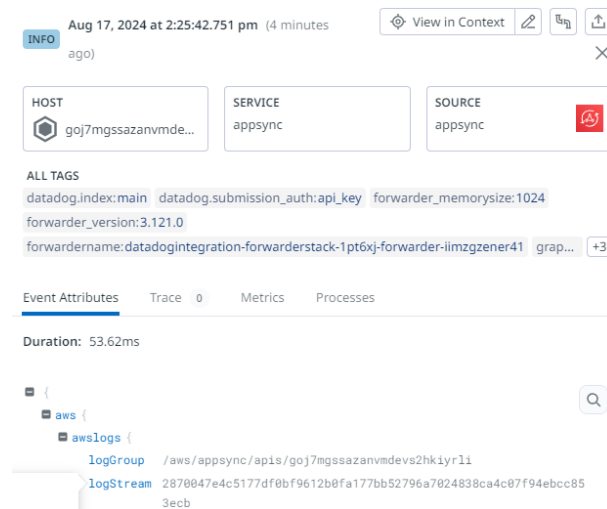
Es sollen nur Logs der API, an der dieses Projekt durchgeführt wird, und von AppSync gesendet wurden in dieses Tortendiagramm einfließen. Das stellen die ersten beiden Filter sicher. Danach wird ein sogenannter Wildcard-Filter auf das Feld “evt.name” angewendet. Das bedeutet, dass alle Logs in dieses Diagramm einfließen, die irgendeinen Wert für dieses Feld haben. Da das Feld nur für Antwort-Logs existiert, werden dadurch alle informativen Logs herausgefiltert, und das gewünschte Diagramm erreicht:

Abbildung 32: Tortendiagramm

Eigene Darstellung

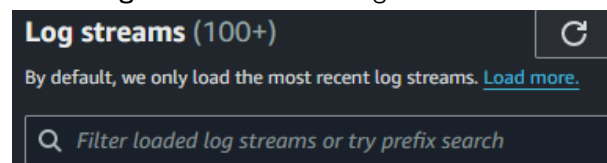
4.4.2 CloudWatch

Aus dem DataDog-Dashboard lässt sich, sollte der Bedarf bestehen, zu jedem Log eines Fehlers der jeweilige logStream herauslesen:

Abbildung 33: LogStream

Eigene Darstellung

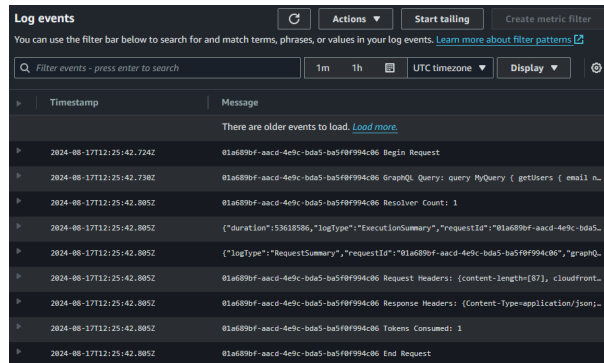
Mit dieser Identifikationsnummer kann dann auf den LogStream in CloudWatch zugegriffen werden, in welchem alle Informationen zu dem Fehler aufzufinden sind. Das ist möglich, indem innerhalb der LogStream-Sammlung der API mithilfe der Identifikationsnummer gefiltert wird:

Abbildung 34: Filter für den LogStream

Eigene Darstellung

Hier ein beispielhafter Ausschnitt aus den Logs:

Abbildung 35: LogStream Beispiel



The screenshot shows the AWS CloudWatch 'Log events' interface. At the top, there are buttons for 'Actions', 'Start tailing', and 'Create metric filter'. Below these is a search bar with the placeholder 'Filter events - press enter to search'. To the right of the search bar are filters for '1m', '1h', 'UTC timezone', and a 'Display' dropdown. The main content area is a table with two columns: 'Timestamp' and 'Message'. The table contains several log events, each with a timestamp and a message. The messages include details about a GraphQL query, resolver count, request headers, response headers, and tokens consumed.

Timestamp	Message
	There are older events to load. Load more
2024-08-17T12:25:42.724Z	01a689bf-aacd-4e9c-bda5-ba5f8f994c06 Begin Request
2024-08-17T12:25:42.738Z	01a689bf-aacd-4e9c-bda5-ba5f8f994c06 GraphQL Query: query MyQuery { getUsers { email n...
2024-08-17T12:25:42.805Z	01a689bf-aacd-4e9c-bda5-ba5f8f994c06 Resolver Count: 1
2024-08-17T12:25:42.805Z	{"duration":13618586,"logType":"ExecutionSummary","requestId":"01a689bf-aacd-4e9c-bda5-...
2024-08-17T12:25:42.805Z	{"logType":"RequestSummary","requestId":"01a689bf-aacd-4e9c-bda5-ba5f8f994c06","graphQ...
2024-08-17T12:25:42.805Z	01a689bf-aacd-4e9c-bda5-ba5f8f994c06 Request Headers: (content-length:[87], cloudfront...
2024-08-17T12:25:42.805Z	01a689bf-aacd-4e9c-bda5-ba5f8f994c06 Response Headers: (Content-Type:application/json;...
2024-08-17T12:25:42.805Z	01a689bf-aacd-4e9c-bda5-ba5f8f994c06 Tokens Consumed: 1
2024-08-17T12:25:42.805Z	01a689bf-aacd-4e9c-bda5-ba5f8f994c06 End Request

Eigene Darstellung

Durch die Informationen, die aus diesen Logs hervorgehen (Fehlertyp, Stelle im Code/in der Abfrage), ist das Debuggen der Fehler erleichtert.

5 Fazit

5.1 Bedeutung für das Unternehmen

Da diese Umsetzung sich auf eine Testumgebung beschränkt, entspringt kein direkter wertschöpfender Nutzen für das Unternehmen aus der Arbeit. Die Infrastruktur, die im Rahmen dieser Praxisarbeit aufgebaut wurde, wird aber, sobald sie auf das Produktiv-System übertragen wird, einen großen positiven Einfluss auf die Maintainability der APIs haben.

5.2 Ausblick

Die Übertragung der im Rahmen dieser Praxisarbeit erarbeiteten Infrastruktur auf das Produktiv-System steht in Zukunft im Mittelpunkt. Entsprechende Anpassungen werden vor bezüglich der Lambda-Funktion und des Terraform-Backend nötig sein, da die Datenstruktur nicht identisch ist. Je nach Feedback der Nutzer sind genauere Fehlerangaben oder Erweiterungen des Datadog-Dashboards möglich.

5.3 Bewertung des Erreichens der Zielstellung

Die Zielsetzungen, die in der Praxisarbeit formuliert wurden, wurden erfüllt. Vor allem im Dashboard wird aber eine Verbesserung nötig sein, um eine bessere Nutzerfreundlichkeit zu garantieren

Anhang

Anhangsverzeichnis

Quellenverzeichnis

Monographien

- Brikman, Y. (2019). *Terraform: Up & Running: Writing Infrastructure as Code*. O'Reilly Media, S. 12.
- Dixon, J. (2018). *Monitoring with Datadog*. O'Reilly Media, S. 25.
- E. Porcello, A. Banks (2018a). *Learning GraphQL*. O'Reilly Media, S. 3.
- E. Porcello, A. Banks (2018b). *Learning GraphQL*. O'Reilly Media, S. 23.
- Foundation, GraphQL (2024). *GraphQL Documentation*. GraphQL Foundation, S. 12.
- Kavis, M. J. (2014). *Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS)*. Wiley, S. 33.
- Keesler, J. Baron H. Bond S. Davis B. Rydahl T. (2017). *AWS Certified Solutions Architect Official Study Guide: Associate Exam*. Sybex, S. 15.
- Masse, Mark (2011). *REST API Design Rulebook*. O'Reilly Media, S. 3.
- Services, Amazon Web (2024a). *Amazon CloudWatch User Guide*. Amazon, S. 7.
- Services, Amazon Web (2024b). *Amazon DynamoDB Developer Guide*. Amazon, S. 22.
- Services, Amazon Web (2024c). *AWS AppSync Developer Guide*. Amazon, S. 12.
- Services, Amazon Web (2024d). *AWS Economics Center - Pricing Overview*. Amazon, S. 7.
- Services, Amazon Web (2024e). *AWS Lambda Developer Guide*. Amazon, S. 11.

Ehrenwörtliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Praxisarbeit selbständig angefertigt habe. Es wurden nur die in der Arbeit ausdrücklich benannten Quellen und Hilfsmittel benutzt. Wörtlich oder sinngemäß übernommenes Gedankengut habe ich als solches kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Langenfeld, 26.08.2024

Thomas Benjamin Hopf