



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	V
Tabellenverzeichnis	VII
1 Einleitung und Motivation	1
1.1 Zielsetzung	1
1.2 Aufbau der Arbeit	1
1.3 Unternehmen	2
1.4 Relevanz für das Unternehmen	2
1.5 Einschränkung	2
2 Grundlagen	4
2.1 Grundlagen der Datenbereitstellung	4
2.2 Grundlagen des Datentransports	4
2.3 Grundlagen des Datenabrufs	4
2.4 Grundlagen des Daten-Monitorings	4
2.5 Konzepte	5
2.5.1 Application Programming Interface (API)	5
2.5.2 Effizienzfaktoren	5
2.5.3 Daten	5
2.6 Werkzeuge	6
2.6.1 AWS	6
2.6.2 Werkzeuge der Datenbereitstellung	6
2.6.3 Werkzeuge des Datentransports	7
2.6.4 Werkzeuge des Datenabrufs	8
2.6.5 Werkzeuge des Daten-Monitorings	8
3 Anforderungen	9
3.1 Datenbereitstellung	9
3.1.1 Erstellung der Datengrundlage	9
3.1.2 Die Test-API	10
3.1.3 Terraform	11
3.2 Datentransport	11

3.2.1	Theorie der Integration	11
3.3	Datenabruf	12
3.3.1	Lambda-Funktion	12
3.4	Daten-Monitoring	12
3.5	DataDog	12
3.5.1	Effizienz	12
3.5.2	Effektivität	13
3.6	CloudWatch	13
4	Umsetzung	14
4.1	Umsetzung der Datenbereitstellung	14
4.1.1	DynamoDB	14
4.1.2	Die API	15
4.1.3	GetUser	18
4.1.4	Terraform	21
4.2	Umsetzung des Datentransports	25
4.2.1	AWS-Datadog-Integration	25
4.3	Umsetzung des Datenabrufs	26
4.4	Umsetzung des Daten-Monitorings	31
4.4.1	DataDog	32
4.4.2	CloudWatch	34
5	Fazit	37
5.1	Bedeutung für das Unternehmen	37
5.2	Bewertung des Erreichens der Zielstellung	37
5.2.1	Erstellung der Datengrundlage	37
5.2.2	Datentransport	37
5.2.3	Datenabruf	37
5.2.4	Daten-Monitoring	38
5.3	Ausblick	38
6	Anhang	39
	Quellenverzeichnis	40
	Ehrenwörtliche Erklärung	41

Abbildungsverzeichnis

Abbildung 1: Beispiel einer JSON-Antwort	7
Abbildung 2: Table Creation	14
Abbildung 3: Datentypen	15
Abbildung 4: Status	15
Abbildung 5: Query und Mutation	16
Abbildung 6: Import der Bibliothek	16
Abbildung 7: Methode Request	17
Abbildung 8: Methode Response	17
Abbildung 9: Alternative Methode Response	18
Abbildung 10: Methoden GetUser	19
Abbildung 11: Methoden GetUsers	20
Abbildung 12: Methoden AddUser	21
Abbildung 13: Definition der Tabelle	22
Abbildung 14: Definition der Berechtigungen	23
Abbildung 15: Definition der Funktions-Resolver	24
Abbildung 16: Aktivierung des Cloud-Watch-Loggings	25
Abbildung 17: GetLogger-Methode	27
Abbildung 18: GetSecret-Methode	27
Abbildung 19: Query-Methode	27
Abbildung 20: Definition der Query	28
Abbildung 21: Formierung des Requests	28
Abbildung 22: Durchführung der Anfrage	28
Abbildung 23: Fehlerhafte Query	29
Abbildung 24: Generieren des API-Keys	29
Abbildung 25: Ausführen der Query	30
Abbildung 26: Generieren des Loggers	30
Abbildung 27: Rückgabe des Felds “errors”	30
Abbildung 28: Korrekte Query	31
Abbildung 29: Rückgabe des korrekten Resultats	31
Abbildung 30: Zeitstrahl; Einstellungen	32
Abbildung 31: Zeitstrahl	33
Abbildung 32: Tortendiagramm Filter	33
Abbildung 33: Tortendiagramm	34

Abbildung 34: LogStream	35
Abbildung 35: Filter für den LogStream	35
Abbildung 36: LogStream Beispiel	36

Tabellenverzeichnis

Tabelle 1: Aufbau der Tabelle	10
---	----

1 Einleitung und Motivation

Die Mitarbeiter der Fachabteilungen benutzen eine in AWS aufgebaute API, um auf betriebsrelevante Daten zuzugreifen. Die Reliabilität dieser API ist dementsprechend betrieblich wichtig. 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. Fehlerdetails werden nicht übermittelt, wodurch die Identifizierung und Behebung von Fehlern 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 Fehlerinformationen transparent gestaltet werden und die Möglichkeit der Überwachung der Response time geschaffen werden. Für die Dokumentation der Fehler soll die Applikation “CloudWatch” Einsicht in die Prozesse geben. Zusätzlich soll die Auftrittshäufigkeit von Fehlern in einem Dashboard dargestellt werden. Die Überwachung der Response time über die Zeit soll durch die Applikation „WatchDog“ erreicht werden. Hierzu sollen relevante Daten zu den jeweiligen Anfragen gesammelt und dargestellt werden, sodass auch Trends erkennbar sind.

1.2 Aufbau der Arbeit

Zuerst sollen die Konzepte und Werkzeuge geklärt werden, mit denen in der Arbeit gearbeitet wird. Dann sollen die konkreten Anforderungen definiert werden, die aus der Zielsetzung hervorgehen. Dann soll die Umsetzung dieser Anforderungen erläutert werden.

1.3 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. Für die Enabling Function ist Effizienz und Effektivität ein wichtiges Kriterium, woraus sich die Relevanz dieser Arbeit ableitet.

1.4 Relevanz für das Unternehmen

Neben den geringeren Prozesskosten in der Verantwortung der Enabling Functions führt eine gesteigerte Effizienz und Effektivität zu einer verbesserten Nutzerfreundlichkeit. Durch Benachrichtigungen können Fehler schneller behoben werden, wodurch die Uptime erhöht wird und durch Monitoring der Response time kann entsprechend bei negativen Trends dieser früher gehandelt werden. Dadurch soll die API schlussendlich, gestützt von den zu implementierenden Anwendungen, zuverlässiger laufen und zu verbessertem Nutzerempfinden führen.

1.5 Einschränkung

Aus zeitlichen Gründen wird im Rahmen dieser Praxisarbeit die Implementierung nur auf einem Test-System durchgeführt, und alle beschriebenen Komponenten sind 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

In diesem Projekt wird ein Testsystem von Grund auf aufgebaut, mit dem das Produktivsystem imitiert werden soll, in das die Infrastruktur später eingebaut werden soll. Das bedeutet, dass auch die Elemente des Produktiv-Systems, die keinen Einfluss auf das Erreichen der Zielstellung haben, im Testsystem repliziert werden müssen, damit das System repräsentativ ist.

Der Aufbau des Testsystems teilt sich in vier Schritte auf:

2.1 Grundlagen der Datenbereitstellung

Im ersten Schritt wird das Produktiv-System nachgebaut. Es muss sichergestellt sein, dass Daten existieren, und dass auf diese Daten zugegriffen werden kann, damit Messungen zu diesem Zugriff stattfinden können.

2.2 Grundlagen des Datentransports

Die Daten, die bei den Messungen erhoben werden, müssen weitergeleitet werden können, damit sie in darauf spezialisierten Programmen verarbeitet werden können.

2.3 Grundlagen des Datenabrufs

Da das Testsystem keine Endnutzer hat, müssen diese simuliert werden, da sonst keine Zugriffsdaten existieren würden.

2.4 Grundlagen des Daten-Monitorings

In der spezialisierten Software müssen die Daten zuletzt noch verarbeitet werden, damit aus ihnen wertvolle Informationen extrahiert werden können.

Um die Umsetzung dieses Plans erklären zu können, sind einige Konzepte und Werkzeuge essentiell.

2.5 Konzepte

Für diese Arbeit relevant sind folgende Konzepte:

2.5.1 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. Fielding, R. T. (2000) 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.

2.5.2 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 (2024b)

2.5.3 Daten

Im Folgenden wird das Konzept der Daten eine zentrale Rolle spielen. Referenziert sind dabei alle solche Daten, mit denen Einsicht in die Gesundheit des Systems ermöglicht werden.

2.6 Werkzeuge

Zur Umsetzung der praktischen Elemente der Praxisarbeit werden die im Folgenden beschriebenen Elemente verwendet:

2.6.1 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. Q. Zhang L. Cheng, R. Boutaba (2010)

2.6.2 Werkzeuge der Datenbereitstellung

Um die Daten bereitzustellen, müssen sie erstellt werden und abrufbar sein. Folgende Werkzeuge werden dafür verwendet:

DynamoDB DynamoDB ist eine vollständig verwaltete Schlüssel-Wert- und Dokumentendatenbank, die bei jeder Größenordnung eine Responsetime von wenigen Millisekunden garantiert. Services, Amazon Web (2024a)

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. **GraphQL2024**

Fehlerbehandlung in GraphQL 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. **GraphQL2024** 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 kein Fehler auf, befindet sich die Antwort auf die Anfrage in dem Feld „data“. **GraphQL2024b** 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.

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

Terraform Terraform ist eine Konfigurationssprache, mit der eine Infrastruktur aufgesetzt wird, die Multi-Cloud-Bereitstellung automatisiert. HashiCorp (2024)

2.6.3 Werkzeuge des Datentransports

Um die Daten transportieren zu können, müssen Integrationen mit den Zielanwendungen hergestellt werden. Folgende Anwendung wird dafür verwendet:

CloudFormation AWS CloudFormation bietet eine Plattform, mit der Integrationen zwischen AWS und externen Anwendungen hergestellt werden können. In Form von sogenannten “Stacks” können in CloudFormation Sammlungen von AWS-Ressourcen gespeichert werden, mit denen Daten von und zu diesen externen Anwendungen fließen können.

2.6.4 Werkzeuge des Datenabrufs

Um Daten automatisiert abrufen zu können, wird die folgende Anwendung verwendet:

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 (2024c)

2.6.5 Werkzeuge des Daten-Monitorings

Folgende Anwendungen werden verwendet, um die erhobenen Daten darzustellen.

DataDog DataDog ist, ähnlich wie CloudWatch, eine Applikation zur 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. DataDog (2024)

CloudWatch CloudWatch 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 und ihre Meldungen dokumentiert, Hinweise und andere Logs gespeichert, und wenn gewünscht Alarmer ausgelöst.

3 Anforderungen

Im Folgenden sollen alle von der Abteilung formulierten Erwartungen, die im Rahmen dieser Praxisarbeit erreicht werden sollen, erläutert werden. Diese Anforderungen teilen sich in die vier Grundaufgaben auf, die in ?? beschrieben wurden.

3.1 Datenbereitstellung

Zuerst müssen die Daten, auf die mithilfe der API zugegriffen wird, erstellt werden und abrufbar gemacht werden.

3.1.1 Erstellung der Datengrundlage

Die Basis der Test-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

3.1.2 Die Test-API

Unter Nutzung dieser Daten soll die Test-API folgende Methoden anbieten. Eine Varianz soll in den Methoden gegeben sein, damit unterschiedliche Laufzeitverhalten und auftretende Fehler simuliert und gemessen werden können.

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.

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.

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. Diese API funktioniert aber nur, wenn das Backend implementiert ist. Dafür ist Terraform zuständig.

3.1.3 Terraform

Das Terraform-Backend soll alle Hintergrundprozesse handhaben, sodass die API ohne manuelle Bearbeitung funktionieren kann. Zu diesen Prozessen gehören die Datenbank-Definition, das Berechtigungs-Management, die Resolver für die jeweiligen Methoden, und die Aktivierung des Loggings. Ist all das implementiert, kann ein Nutzer im Testsystem auf Daten zugreifen, die dabei entstehenden Informationen können jedoch noch nicht vollständig verarbeitet werden, da die Daten noch nicht überallhin verteilt werden können:

3.2 Datentransport

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.

3.2.1 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.

3.3 Datenabruf

3.3.1 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.

3.4 Daten-Monitoring

Im Rahmen des Daten-Monitorings sollen zwei verschiedene Erkenntnisse gesammelt werden: Zum einen soll ermittelt werden, ob die API eine stabile Response time hat, und zum anderen sollen Fehlerquellen ermittelt und ausgewertet werden. Ersteres soll in DataDog umgesetzt werden.

3.5 DataDog

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.

Die Auswertung von Fehlern soll über CloudWatch ablaufen.

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 aus.

4 Umsetzung

Nun müssen die Anforderungen im Testsystem umgesetzt werden. Die Aufteilung bleibt dabei gleich, demnach ist der erste Schritt die Umsetzung der Datenbereitstellung.

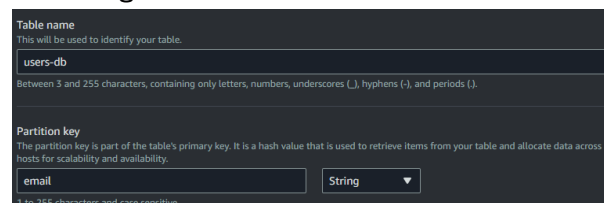
4.1 Umsetzung der Datenbereitstellung

Bevor die Daten für Nutzer bereitgestellt werden können, müssen sie erstellt werden. Dafür wird die Tabelle erstellt und befüllt.

4.1.1 DynamoDB

DynamoDB wird verwendet, um die Datengrundlage zu garantieren. In DynamoDB wird eine Tabelle erstellt die, wie in dem Schema aus 1 vorgegeben, die E-Mail als Schlüssel aufweist:

Abbildung 2: Table Creation



The screenshot shows the AWS DynamoDB 'Table Creation' interface. It has a dark theme. The 'Table name' section is at the top, with a text input field containing 'users-db'. Below it, a note states: 'Between 3 and 255 characters, containing only letters, numbers, underscores (_), hyphens (-), and periods (.)'.

The 'Partition key' section is below that. It includes a text input field containing 'email' and a dropdown menu set to 'String'. A note below the input field states: '1 to 255 characters and case sensitive.'

Eigene Darstellung

Diese Tabelle sollte danach noch befüllt werden, damit Daten durch die API ausgegeben werden können. Dabei ist als einziges das Schema wichtig, da der Inhalt für die Simulation nicht relevant ist. Jetzt kann die API erstellt werden, mit der die Daten für einen Nutzer (in diesem Fall für die Lambda-Funktion) erreichbar gemacht werden.

4.1.2 Die API

Der erste Schritt ist die Definition des Datenschema und der Methoden in GraphQL.

GraphQL Der GraphQL-Code erfüllt zwei verschiedene Aufgaben **GraphQL2024**: 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 3: 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 4: 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 5: Query und Mutation

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

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

Eigene Darstellung

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 6: 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 7: 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 8: 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 9: Alternative Methode Response

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

Eigene Darstellung

4.1.3 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 10: 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 Eingabeparameter nur die Möglichkeit bestehen muss, die auszugebenden Felder auszuwählen. Für die Ausgabe muss aber definiert werden, welche Operation ausgeführt wird,

Die resultierende Antwort ist die Rückgabe der Abfrage:

Abbildung 11: 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 12: 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. Damit aber die API auch abrufbar ist, muss das Backend gehandhabt sein. Dafür ist Terraform zuständig.

4.1.4 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 13: 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 14: 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

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

die Methoden ausgeführt werden können. Der Resolver wird dann als Resource abgespeichert. Beispielhaft ist hier die Methode AddUser dargestellt:

Abbildung 15: 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 16: Aktivierung des Cloud-Watch-Loggings

```

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

```

Eigene Darstellung

Damit ist der Abruf von Daten gewährleistet. Nun muss das Verteilen von Daten ermöglicht werden.

4.2 Umsetzung des Datentransports

Damit nun erhobenen Daten über die erstellte API verteilt werden können, muss die Infrastruktur für die Verteilung der Daten aufgebaut werden:

4.2.1 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.

4.3 Umsetzung des Datenabrufs

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 zweite 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 17: 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 18: 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 19: Query-Methode

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

Eigene Darstellung

Die Bearbeitung ist in mehrere Schritte gegliedert:

1. Definition der query:

Abbildung 20: 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 21: 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 22: 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 23: 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 24: 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 25: 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 26: Generieren des Loggers

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

Eigene Darstellung

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

Abbildung 27: 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 28: 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 29: Rückgabe des korrekten Resultats

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

Eigene Darstellung

Jetzt ist ein Datenabruf in externen Anwendungen möglich. Der letzte Schritt ist jetzt, die Daten in ebendiesen Anwendungen auszuwerten.

4.4 Umsetzung des Daten-Monitorings

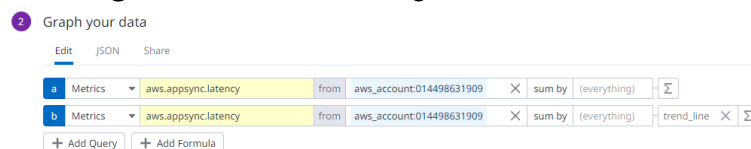
Zuerst soll die Stabilität der API kontrolliert werden. Das soll über ein DataDog-Dashboard geschehen.

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 aber auch ein Zeitstrahl für Tendenzen wären denkbar. Die Auswahl basiert hier nicht darauf, welches Diagramm am besten für den Nutzungsfall geeignet ist. Schließlich ist diese Entscheidung auch von Nutzerpräferenzen abhängig, und kann aufgrund des frühen Entwicklungsstadiums noch nicht miteinbezogen werden kann.

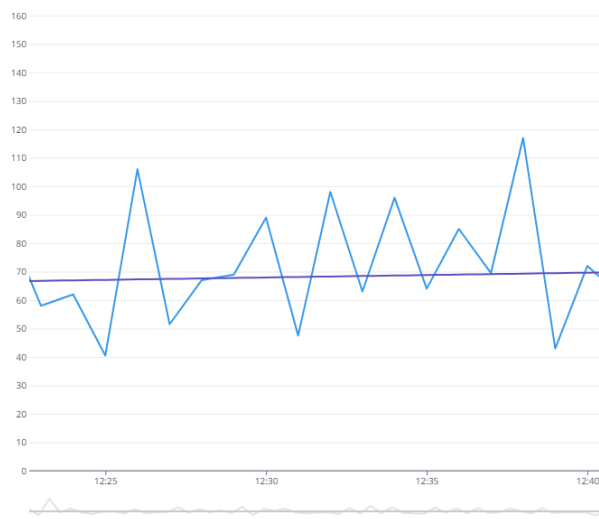
Zeitstrahl Für den Zeitstrahl wird die Metrik “aws.appsync.latency” verwendet. Ein Strahl stellt dabei die exakte Response time zu einem bestimmten Zeitpunkt dar, während der andere die zugehörige Trendlinie bildet, die das Visualisieren von Trends vereinfachen soll:

Abbildung 30: Zeitstrahl; Einstellungen

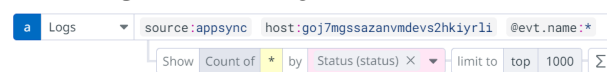


Eigene Darstellung

Hier ein beispielhafter Ausschnitt des Graphen:

Abbildung 31: 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 32: Tortendiagramm Filter*Eigene Darstellung*

Es sollen nur Logs der API, an der dieses Projekt durchgeführt wird, und von AppSync gesendet wurden in dieses Tortendiagram 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:

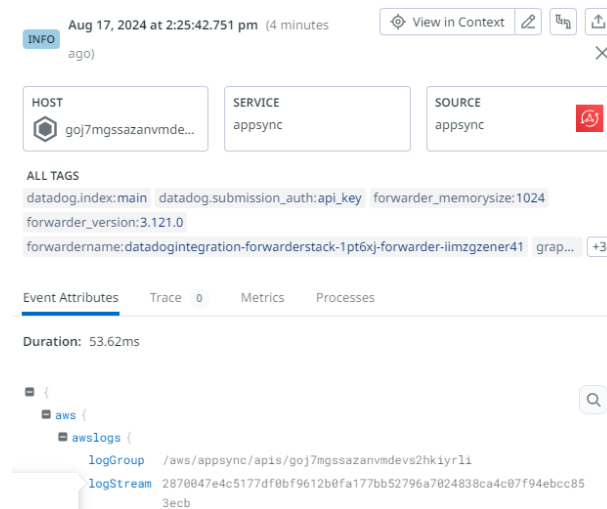
Abbildung 33: Tortendiagramm



Eigene Darstellung

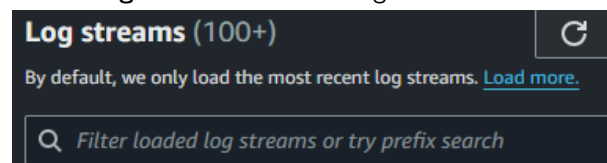
4.4.2 CloudWatch

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

Abbildung 34: LogStream

Eigene Darstellung

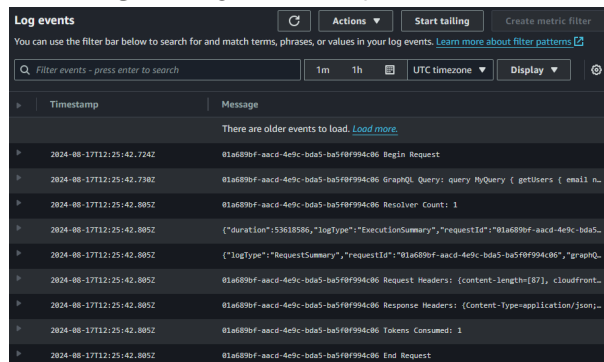
Mit dieser Identifikationsnummer kann dann auf den LogStream in CloudWatch zugegriffen werden, in welchem alle Informationen zu dem Fehler aufzufinden sind. Das ist möglich, indem innerhalb der LogStream-Sammlung der API mithilfe der Identifikationsnummer gefiltert wird:

Abbildung 35: Filter für den LogStream

Eigene Darstellung

Hier ein beispielhafter Ausschnitt aus den Logs:

Abbildung 36: LogStream Beispiel



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 aus der Arbeit für das Unternehmen kein direkter wertschöpfender Nutzen. 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 Bewertung des Erreichens der Zielstellung

Um das Erreichen der Zielstellung zu evaluieren, werden im Folgenden die vier in den Anforderungen definierten Erwartungen bewertet.

5.2.1 Erstellung der Datengrundlage

Die Datengrundlage ist in Form der DynamoDB-Tabelle vollständig funktionsfähig erstellt. Ebenso vollständig funktionsfähig ist die API und das zugehörige Terraform-Backend. Alle angeforderten Funktionen sind hier vollständig abgedeckt

5.2.2 Datentransport

Die für den Datentransport benötigte Infrastruktur ist vollständig aufgebaut; Daten können zwischen allen Anwendungen fließen, bei denen ein Datenfluss vorgesehen war.

5.2.3 Datenabruf

Die Lambda-Funktionen, mit denen API-Anfragen automatisch generiert werden, funktionieren fehlerfrei.

5.2.4 Daten-Monitoring

Das DataDog-Dashboard enthält alle in den Anforderungen vorgesehene Funktionen. Insofern sind die Anforderungen erfüllt, wenngleich weitere Entwicklung mit Einbezug von Nutzerfeedback nötig ist, um das Dashboard als vollständig werten zu können.

Da in CloudWatch keine zusätzliche Entwicklung nötig war, ist die Funktionsweise vorhersehbar. In den Anforderungen wurden nur Funktionen beschrieben, die von CloudWatch abgedeckt werden können.

Insgesamt wurde, gemäß der Zielstellung, eine Infrastruktur erstellt, mit der die Effizienz und Effektivität einer API im AWS-System verbessert werden kann.

5.3 Ausblick

Die Übertragung der im Rahmen dieser Praxisarbeit erarbeiteten Infrastruktur auf das Produktiv-System steht in Zukunft im Mittelpunkt. Entsprechende Anpassungen werden vor allem bezüglich der Lambda-Funktion und des Terraform-Backends nötig sein, da die Datenstruktur nicht identisch ist. Erweiterungen der bereits erstellten Infrastruktur können je nach Nutzer-Rückmeldungen durchgeführt werden; erwartet sind diese hauptsächlich im DataDog-Dashboard.

Quellenverzeichnis

Monographien

DataDog (2024). *Getting Started with Datadog*. DataDog.

E. Porcello, A. Banks (2018a). *Learning GraphQL*. O'Reilly Media.

E. Porcello, A. Banks (2018b). *Learning GraphQL*. O'Reilly Media.

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. Doctoral Dissertation. University of California.

HashiCorp (2024). *Terraform Documentation*. HashiCorp.

Kavis, M. J. (2014). *Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS)*. Wiley, S. 37.

Q. Zhang L. Cheng, R. Boutaba (2010). *Cloud Computing: State-of-the-art and research challenges*. Journal of Internet Services und Applications.

Services, Amazon Web (2024a). *Amazon DynamoDB Developer Guide*. Amazon.

Services, Amazon Web (2024b). *AWS Economics Center - Pricing Overview*. Amazon.

Services, Amazon Web (2024c). *AWS Lambda Developer Guide*. Amazon.

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