I Inhaltsverzeichnis

II Abbildungsverzeichnis

1 Einleitung

Die dotSource GmbH setzt für ihre Kunden verschiedenste Projekte im eCommerce-Bereich um. Dazu werden moderne Ansätze, wie agiles Projektmanagement und cloud-native Entwicklung mit Microservices verfolgt. Dabei ist die dotSource GmbH bestrebt neue Technologien und Trends im Interesse ihrer Kunden zu erforschen und umzusetzen.

Daher wird in einem Kundenprojekt auf Functions as a Service gesetzt. In dieser Arbeit sollen als erstes die Begriffe „Functions as a Service“ und „Middleware“ erklärt und dann das Kundenprojekt unter verschiedenen Gesichtspunkten analysiert werden. Zudem sollen einige Anforderungen für das Projekt ermittelt und formuliert werden. Dann werden einige existierende Ansätze betrachtet und erläutert, warum diese nicht eingesetzt werden. Darauffolgend soll ein Framework entwickelt werden, welches die gestellten Anforderungen umsetzen kann. Dieses Framework kann dann in das Projekt integriert und von zukünftigen Projekten mit ähnlichen Anforderungen verwendet werden. Zuletzt soll das Framework in einem Teil des Kundenprojekts getestet und dessen Auswirkungen untersucht werden. Dabei soll auch überprüft werden, ob die Anforderungen tatsächlich umgesetzt wurden. Dies soll dann als Grundlage dafür dienen, um zu entscheiden, ob das Framework dann in das Kundenprojekt integriert werden soll.

2 Functions as a Service und Middleware

2.1 Functions as a Service

Functions as a Service (FaaS) ist ein, sich immer weiterverbreitendes Modell, um Anwendungen in der Cloud bereitzustellen. Mehrere Cloudanbieter bieten ihre eigenen FaaS-Dienste an. Nennenswert sind hierbei Azure Functions[1], AWS Lambda[2], Google Cloud Functions[3].

Ursprünglich wurden Anwendung als monolithische Systeme entwickelt, in welchen alle Komponenten enthalten waren. Aktuell ist jedoch der Trend diese Systeme in kleine unabhängige Komponente, sogenannte Microservices, zu unterteilen. FaaS folgt dem Trend Anwendungen in einzelne kleine Komponenten zu unterteilen.[4] Wie in Abbildung 1 zu sehen, kann FaaS als der nächste Schritt nach dem Microservices-Modell angesehen werden.

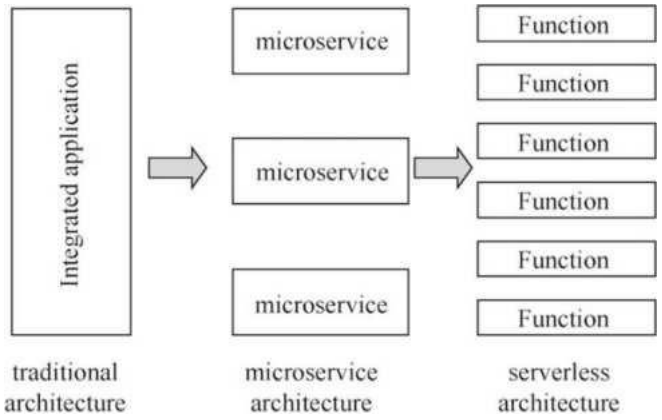

Abbildung 1: Kleinere Komponenten mit FaaS
Quelle: [Hua23]

Ähnlich wie bei den Microservices ermöglicht FaaS, die einzelnen Komponenten unabhängig voneinander zu entwickeln und betreiben. Komponenten sind hier jedoch in einer Programmiersprache definierte Funktionen. FaaS unterscheidet sich zudem

[1] [Mic24].
[2] [Ama24].
[3] [Goo24].
[4] [Hua23], S. 366.

von den Microservices insbesondere in der Ausführungsdauer. So werden FaaS-Instanzen nicht dauerhaft betrieben, sondern werden nur als Reaktion auf festgelegte Ereignisse gestartet. Solche Ereignisse können ein Datei-Upload, eine neue Message in einer Message-Queue, oder ein HTTP-Aufruf sein. Damit zusammenhängend berechnen Cloudanbieter FaaS-Dienste häufig pro Funktionsaufruf.[5]

Außerdem folgt FaaS dem Serverless-Ansatz. So werden dem Entwickler Aufgaben, wie der Verwaltung der Server und Skalierung de Anwendung abgenommen. Diese werden nämlich vom Cloudanbieter übernommen. So können Entwickler sich auf die eigentliche Entwicklung konzentrieren.[6]

Jedoch bringt FaaS auch einige Nachteile. Da die einzelnen Funktionen über eine Netzwerkschnittstelle kommunizieren entstehen zusätzliche Latenzen. Außerdem sind die Funktionen zustandslos. Muss Zustand über mehrere Aufrufe erhalten werden, so wird ein externer Speicher benötigt. Ein weiterer Nachteil ist auch Kaltstarts. Diese erscheinen, wenn eine Funktion über einen längeren Zeitraum nicht aufgerufen, und deswegen vom Cloudanbieter herunterskaliert wurde. Wird die Funktion wieder aufgerufen, so muss diese erst hochgefahren werden.

2.2 Middleware-Konzept

Middleware sind Programme, welche im Hintergrund laufen, und zwischen der Netzwerkschicht und der Anwendungsschicht liegen. Middleware bietet der Anwendungsschicht verschiedene Dienste an, um die Komplexität und Infrastruktur der Kommunikation zu verbergen.[7] Sie ermöglicht die Kommunikation zwischen den Anwendungen und erleichtert die Verwaltung und Überwachung des Systems. Wie in Abbildung 2 zu sehen, kann Middleware daher als Plattform betrachtet werden, auf der die Anwendungen entwickelt und betrieben werden können. Hier verbindet die Middleware verschiedene Anwendungen über mehrere Maschinen hinweg.

[5] Vgl. ebenda, S. 367.
[6] Vgl. ebenda, S. 365.
[7] [GK24].

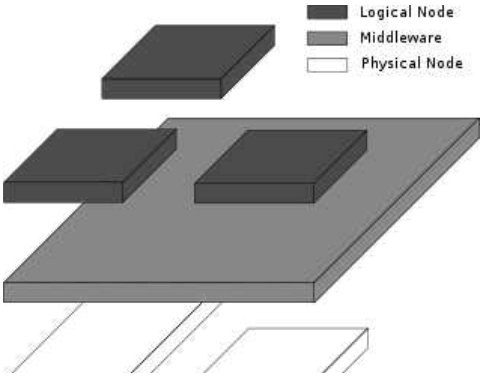

Abbildung 2: Middleware als Plattform
Quelle: [Wik24]

Wenn zum Beispiel ein Legacy-System und ein modernes System verknüpft werden sollen, kann Middleware eingesetzt werden, um zwischen den beiden Systemen zu vermitteln. Sie kann aber auch zwischen einem Client und einem Server geschaltet werden, um dort eingehende Anfragen zu analysieren, gegeben falls zu bearbeiten und passende Antworten zurückzugeben.
Wie in Abbildung 3 zu sehen kann Middleware in verschiedene Arten eingeteilt werden.

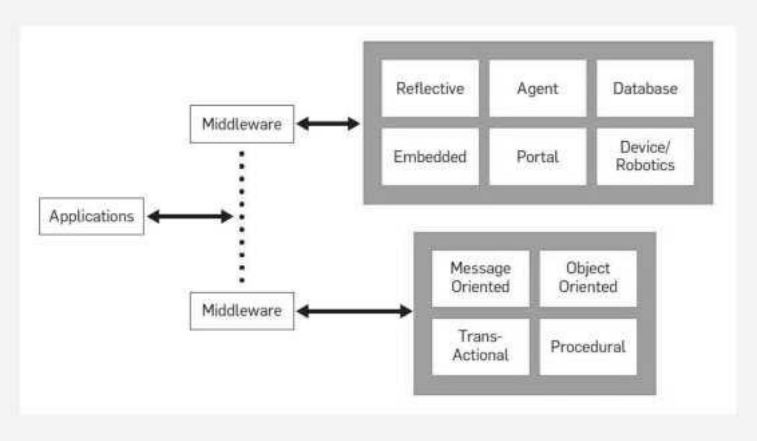

Abbildung 3: Arten von Middleware
Quelle: [GK24]

Dabei kann man auf verschiedene Weisen kategorisieren, einmal nach der Architektur und einmal nach der Anwendung. Bei der Kategorisierung nach der Architektur lässt sich Middleware in Nachrichtenorientierte, objektorientierte, transaktionale, und prozedurale Middleware einteilen. Bei der anwendungsbasierten Kategorisierung kann

man Middleware in reflektive, Agenten-, Datenbank-, eingebettete, Portal-, und Robotik-Middleware unterteilen.

Middleware ist für ein breites Spektrum an verschiedenen Aufgaben verantwortlich, und tritt daher in verschiedenen Formen auf: „Middleware kann Anwendungs-Runtimes, die Integration von Unternehmensanwendungen und verschiedene Arten von Cloud-Services umfassen. Datenverwaltung, Anwendungsservices, Messaging, Authentifizierung und API-Management werden üblicherweise von der Middleware gehandhabt.“[8]

In Express, einem Framework für die Entwicklung von Webservern in Node.js, können vom Entwickler eigene Middleware geschrieben und verwendet werden.[9] Diese analysieren und bearbeiten eingehende HTTP-Anfragen, bevor diese an den Endpunkt zur eigentlichen Verarbeitung gereicht werden. Dies ermöglicht das Auslagern von Funktionalitäten, wie das Laden von Daten aus einer Datenbank oder die Authentifizierung einer Anfrage. In Abbildung 4 ist zu sehen, wie eine Anfrage von einer Middleware zur nächsten durchgereicht wird, bis sie am Ende beim eigentlichen Route-Handler ankommt.

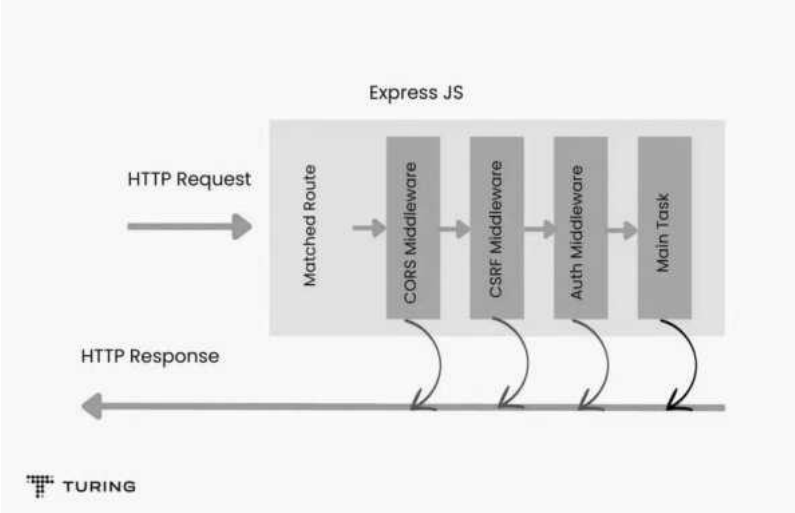

Abbildung 4: Middleware in Express
Quelle: [Tur24]

[8] [Red24].
[9] [Exp24].

Dabei erfüllt jede Middleware eine bestimmte Aufgabe. Außerdem fällt auf, dass die Middlewares verfrüht eine Antwort zurückgeben können, und die Anfrage dann nicht mehr weitergereicht wird. Dies ermöglicht der Auth-Middleware eine Anfrage nur zum Route-Handler weiterzuleiten, wenn diese von einem authentifizierten Nutzer stammt. Zudem ermöglicht Express das Definieren und Verwenden von Middleware zur Fehlerbehandlung.[10]

[10] [Exp24].

3 Analyse eines Kundenprojekts

In diesem Kapitel soll ein Projekt für einen Kunden der dotSource GmbH analysiert werden. Dabei sollen die verwendeten Laufzeitumgebungen, Programmiersprachen und Frameworks betrachtet werden. Zudem sollen einige Besonderheiten aufgezeigt werden.

Als Programmiersprache und Laufzeitumgebung werden JavaScript und Node.js verwendet. Node.js ermöglicht die serverseitige Verwendung von JavaScript, um zum Beispiel Webanwendungen zu erstellen.

Das Projekt wurde für den Einsatz mit AWS-Lambda entwickelt. Dies ermöglicht eine schnelle Entwicklung und einen einfachen Betrieb in der Cloud, da sich die Entwickler mehr auf den Anwendungscode konzentrieren müssen und sich nicht mehr mit dem Betrieb und der Skalierung der Server beschäftigen müssen. Jedoch wurde der Code für den Betrieb mit AWS Lambda geschrieben und kann nur mit Änderungen auf anderen FaaS-Angeboten ausgeführt werden. Dies erschwert eine zukünftige Migration zu einem anderen Cloudanbieter.

Außerdem wird, wie in Abbildung 5 verdeutlicht, pro Endpunkt eine eigene AWS-Lambda-Applikation erstellt. Dadurch ist die Verarbeitung für Anfragen mit unterschiedlichen HTTP-Methoden in verschiedene Applikationen aufgeteilt.

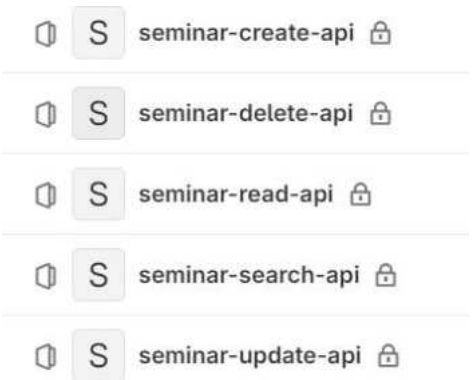

Abbildung 5: Projektstruktur für Endpunkte

Das Projekt lagert bereits Funktionalitäten in asynchrone JavaScript-Funktionen aus. Wie in Abbildung 6 zu sehen, werden die Funktionen im AWS-Lambda-Handler mittels einer sogenannten Promise-Chain verknüpft. Dabei werden diese mittels der „then“-Methode verknüpft. Auftretende Fehler werden in einer Funktion, welche der „catch“-Methode übergeben wird abgefangen und behandelt.

```
  return Promise.resolve()
    .then(() => readConfiguredRoles(redisOptions, event.requestContext.stage))
    .then(() => prepareValues(event))
    .then((values) => readConfig(values))
    .then((values) => validateRequestParameters(values))
    .then((values) => tryGetCustomer(values))
    .then((values) => checkPermissions(values))
    .then((values) => executeSeminarRequest(values))
    .then(responses.ok)
    .then(appResolve())
    .catch(appReject());
};
```

Abbildung 6: Verknüpfung per Promisechain

4 Anforderungen

Das Kundenprojekt enthält jedoch einige Nachteile, welche korrigiert werden sollen. In diesem Kapitel werden dazu einige Anforderungen an das Kundenprojekt gestellt.

4.1 Auslagerung von Nicht-Geschäftslogik

Das Projekt verwendet zur Auslagerung von Code Promise-Chains. Die Verwendung einer Promise-Chain birgt jedoch auch einige Nachteile. So wird bei deren Ausführung jede ihrer Funktionen ausgeführt. Hierzu soll eine Alternative gefunden werden. Mit dieser soll es möglich sein früher eine Antwort zurückzugeben.

4.2 Unabhängigkeit vom Cloudanbieter

Unter Umständen ist zu einem späteren Zeitpunkt eine Migration zu einem anderen Cloudanbieter und dessen FaaS-Angebot erwünscht. Dazu wären jedoch Änderungen am bestehenden Code nötig. Dies wird bei großen Projekten sehr aufwendig. Stattdessen soll das Projekt mit möglichst wenigen Änderungen im Code zu einem anderen Cloudanbieter zu migrieren sein.

4.3 Routing nach HTTP-Anfragemethoden

Es soll außerdem möglich sein, mehrere Endpunkte mit gleichem Pfad, jedoch unterschiedlichen HTTP-Methoden, in einer FaaS-Funktion zusammenzufassen. Wie in Abbildung 7 zu sehen muss dazu in zwischen den HTTP-Methoden unterschieden, und der richtige Endpunkt aufgerufen werden.

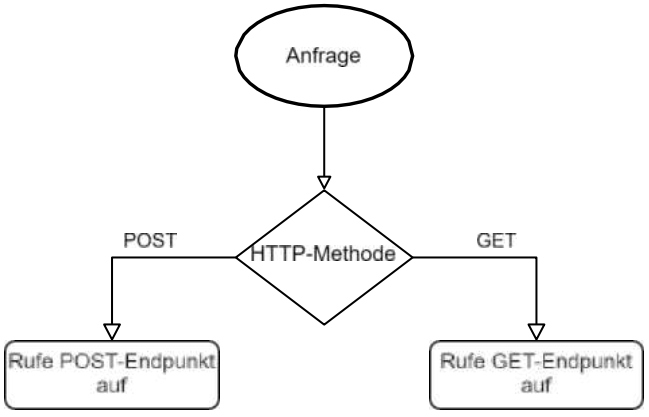

Abbildung 7: Routing nach HTTP-Methoden

5 Existierende Ansätze

Es existieren bereits einige Projekte, welche jeweils die Anforderungen teilweise erfüllen. Nachfolgend sollen diese analysiert und erklärt werden, warum diese nicht verwendet wurden.

5.1 Azure Middleware Engine

Als erstes soll die Azure Middleware Engine von Emanuel Casco betrachtet werden. Es ermöglicht die Verwendung des Middleware-Patterns in Azure Functions. Das Framework bietet dafür unterschiedliche Funktionen an:
Zunächst wird die Funktion “use” angeboten. Mit dieser kann ähnlich wie in Express eine Middleware registriert werden. Middleware werden in der Reihenfolge ausgeführt, in der sie registriert wurden. Mit “useIf” kann eine Middleware registriert werden, welche nur unter bestimmten Bedingungen ausgeführt werden soll. Um über ein Array von Elementen zu iterieren wird die Funktion “iterate” angeboten. Zudem ist in das Framework Validierung mit der Bibliothek joi eingebaut. Dies kann mittels “validate” und einem joi-Schemas verwendet werden. Zuletzt können mit “catch” Funktionen zur Fehlerbehandlung registriert werden. Sollte während der Ausführung des Middlewarestacks ein Fehler geworfen werden, so kann dieser hier behandelt werden. Wie zu sehen war, bietet Azure Middleware Engine nicht nur die Möglichkeit, Middleware in Azure Functions zu verwenden, sondern auch weitere Funktionalitäten an.[11]
Jedoch wurde es speziell für Azure Functions erschaffen und wird zudem nicht mehr aktiv entwickelt. Daher ist es nicht mit der aktuellen Version von Azure Functions und, mit den FaaS-Angeboten anderer Cloud-Anbieter kompatibel.

5.2 Azure-Function-Express

Azure-Function-Express ermöglicht die Einbettung einer Express-Applikation in eine Azure Function. Dadurch können von Express gewohnte Middleware einfach

[11] [Cas24].

verwendet werden. Das Framework bietet zudem das gewohnte Routing nach Pfaden an, und unterscheidet dabei auch zwischen unterschiedlen HTTP-Methoden.
Hiermit sind die Kriterien Unterstützung des Middleware-Patterns und Routing nach HTTP-Methoden zwar erfüllt. Damit diese Bibliothek jedoch in Betracht gezogen werden kann müsste sie auch AWS Lambda und Google Cloud Functions unterstützen. Zudem unterstützt sie nicht die neueste Version von Azure Functions.[12]

5.3 Serverless NestJS

NestJS ist ein Framework mit dem Web-Anwendungen entwickelt werden können. Hierbei wird eine Applikation aus verschiedenen Module zusammengebaut. Diese Module enthalten Controller. Welche Anfragen verarbeiten, und Provider, welche den Controllern und anderen Providern Dienste zur Verfügung stellen.
NestJS ermöglicht die Auslagerung von Code in verschiedene Klassen von Komponenten, welche unterschiedliche Aufgaben erfüllen. So gibt zunächst Middleware, welche keine spezifischen Aufgaben erfüllt. Dann gibt es Exception filters. In diesen können geworfene Fehler behandelt werden. In Pipes können Eingaben in den Anfragen validiert und geparst, oder Anfragen anderweitig transformiert werden. Um Autorisierung und Authentifizierung umzusetzen können Guards eingesetzt werden. Als letztes gibt es Interceptors. Diese können Anfragen oder Antworten abfangen, um weitere Funktionalitäten bereitzustellen.
NestJS bietet keine direkte Unterstützung für die FaaS-Angebote der Cloudanbieter. Um also NestJS in einem FaaS-Angebot zu verwenden, muss zunächst NestJS manuell integriert werden.[13]

5.4 Modofun

Modofun steht für Modular Functions und ist ein Framework für serverlose Node.js-Anwendungen. Es unterstützt unter anderem das Routen zu mehreren Funktionen, das Parsen von Parametern und Middleware.

---

[12] [Mer24].
[13] [Doc24].

Es kann Express kompatible Middleware verwendet werden. Dadurch steht ein großes Arsenal an bereits existierender Middleware zur Verfügung. Zudem unterstützt modofun die FaaS-Angebote AWS Lambda, Azure Functions, und Google Cloud Functions.[14]

Modofun unterstützt jedoch nicht das Routen nach HTTP-Methoden und wird außerdem nicht mehr weiterentwickelt.

[14] [Fpt19].

6 Umsetzung

Um die Anforderungen umzusetzen, soll ein Framework entwickelt werden, welches im Kundenprojekt verwendet werden kann. Die Entwicklung eines Frameworks bietet den Vorteil, dass auch zukünftige Projekte mit ähnlichen Anforderungen auf dieses zurückgreifen können.

6.1 Auslagerung von Nicht-Geschäftslogik

Um Code auszulagern, kommt das Middleware-Pattern in Betracht. Diese soll sich ähnlich wie bei Express verhalten. Dabei kann der Entwickler eigene Middleware erstellen, welche eingehende Anfragen verarbeiten und ausgehende Antworten überarbeiten kann. In diese Middlewares können verschiedene Aufgaben ausgelagert werden. Diese reichen von Logging der Anfragen, Caching, Validierung und Parsen von Parametern bis zur Authentifizierung und Autorisierung von Anfragen und Nutzern. Es generische Middleware geben, welche für alle Aufrufe ausgeführt wird, und Middleware, welche nur für Aufrufe mit bestimmten HTTP-Methoden ausgeführt wird. Zudem soll es wie bei Express möglich sein, Middleware für die Fehlerbehandlung zu definieren. Die Middlewares sollen mittels des Builder-Patterns registrierbar sein.
Abbildung 8 zeigt, wie die Middlewares bei einem Aufruf aufgerufen werden. Bei einem Aufruf wird zunächst die generische Middleware ausgeführt, dann wird die zur HTTP-Methode passende spezifische Middleware ausgeführt, und als letztes der Handler für den Endpunkt.

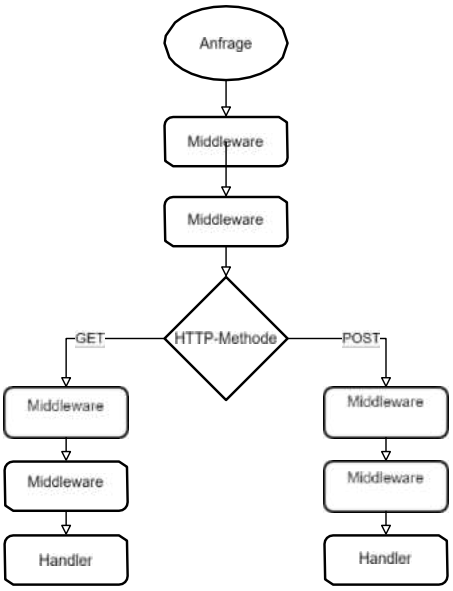

Abbildung 8: Ablauf der Aufrufe der Middlewares und Handler

Die Fehlerbehandlungsmiddleware wird ausgeführt, wenn bei den anderen Middlewares oder dem Endpunkthandler ein Fehler geworfen wurde.

Um die Middlewares auszuführen, wird eine next-Funktion erstellt, welche von den Middlewares aufgerufen werden kann. Diese ruft dann die nächste Middleware im Middleware-Stack auf, oder wenn es keine weitere Middleware gibt, den Handler für den Endpunkt. Dabei überprüft sie, ob eine Middleware oder ein Handler einen Fehler geworfen hat, und führt dann gegeben falls den Middleware-Stack für die Fehlerbehandlung aus.

Den Middlewares wird das Anfrageobjekt, ein Loggerobjekt, und die next-Funktion übergeben. Aus dem Anfrageobjekt können benötigte Informationen gelesen werden. Zudem können im Anfrageobjekt neue Informationen für die nachfolgenden Middlewares und dem Handler gespeichert werden. Das Loggerobjekt kann verwendet werden, um Informationen über den Betrieb festzuhalten oder Warnungen und Fehler auszugeben. Eine Middleware kann die Verarbeitung einer Anfrage frühzeitig unterbrechen, indem sie die next-Funktion nicht aufruft, und stattdessen direkt eine Antwort zurückgibt.

6.2 Unabhängigkeit vom Cloudanbieter

Um die Migration zu einem anderen Cloudanbieter zu vereinfachen, werden eigene Schemas der Anfrage- und Antwortobjekte erstellt. Bei einem Aufruf werden die Anfrageobjekte des Cloudanbieters zur Verarbeitung in das eigene Schema umgewandelt. Nach der Verarbeitung des Aufrufs werden die Antwortobjekte in das Schema des Cloudanbieters konvertiert. Als Letztes werden die Antworten im Schema des Cloudanbieters dann zurückgegeben.

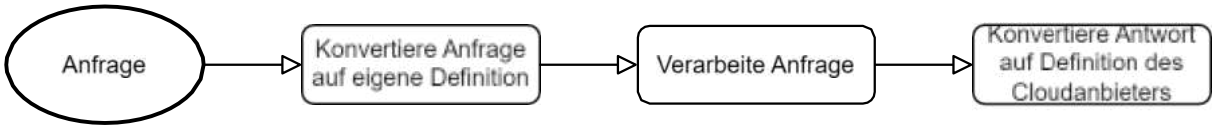

Abbildung 9: Konvertierung der Anfrage- und Antwortobjekte

In Abbildung 10 ist ein Beispiel einer Anfrage im eigenen Schema zu sehen. Die Anfrage enthält dann die HTTP-Methode, den Pfad, gegeben falls Parameter aus der URL-Query und dem Pfad, des Headers und den Body. Über „data“ können Informationen von vorherigen Middlewares und neue an nachfolgende Middlewares und Handler gegeben werden.

Das Framework muss sicherstellen, dass die Anwendung auf diese Daten im gleichen Schema zugreifen kann. Jedoch unterstützt zum Beispiel Google Cloud Functions keine Pfadparameter. Diese Funktionalität muss daher vom Framework umgesetzt werden.

```
{
    "_method": "GET",
    "_path": "/hello",
    "_query": { "name": "Ben" },
    "_params": { "age": "20" },
    "_headers": {
        "Accept": "application/json, text/plain, */*",
        "Accept-Encoding": "gzip, compress, deflate, br",
        "Connection": "close",
        "Host": "127.0.0.1:3000",
        "Request-Start-Time": "1707145217732",
        "User-Agent": "axios/1.5.1"
    },
    "_body": null,
    "data": {}
}
```

Abbildung 10: Beispiel einer Anfrage

Nachdem die Anfrage von der Anwendung verarbeitet wurde, wird dem Framework ein Antwortobjekt zurückgegeben. Dieses enthält, wie in Abbildung 11, zu sehen den Status-Code, die Headers und den Body, welche dem Client übergeben werden sollen.

```
{
    "statusCode": 200,
    "headers": { "Content-Type": "text/html" },
    "body": "<h1>Hello Ben. You are 20 years old!</h1>"
}
```

Abbildung 11: Beispiel eines Antwortobjekts

6.3 Routing nach HTTP-Anfragemethoden

Das Routing nach HTTP-Methoden ermöglicht es mehrere Endpunkte in einer FaaS-Funktion zusammenzufassen. Dadurch wird es erleichtert Middleware für verschiedene Endpunkte zu verwenden, solange die Endpunkte den gleichen Pfad besitzen. Um die Endpunkte den HTTP-Methoden zuzuordnen wird die in Node.js eingebaute Klasse „Map“ verwendet. Bei einem Aufruf werden in der Map die Middlewares und der Endpunkt nach der in der Anfrage angegebenen HTTP-Methode ausgelesen und anschließend aufgerufen.

7 Analyse der Umsetzung

In diesem Kapitel soll die Umsetzung analysiert werden. Dabei soll überprüft werden, ob die Anforderungen erfüllt wurden.

Dazu sollten einige Lambda-Funktionen des Kundenprojekts für die Verwendung mit dem Framework umgeschrieben und auf AWS ausgeführt werden. Jedoch konnte Aufgrund der Komplexität des Projekts und aus Zeitgründen im Rahmen der Projektarbeit nur eine Lambda-Funktion umgeschrieben werden. Um die testen, ob das Framework die Anforderungen an die Cloudunabhängigkeit und dem HTTP-Methoden-Routing erfüllt, wurde eine einfache Beispielanwendung erstellt.

7.1 Auslagerung von Code

Wie in Abbildung 13 zu sehen wurden im Kundenprojekt Funktionalitäten in Funktionen ausgelagert, welche mittels Promise-Chains verknüpft wurden. Diese Funktionen sind in Abbildung 12 nun als Middlewares deklariert und in unterschiedle Typen (generische, spezifische, Fehlerbehandlung) aufgeteilt. Zusätzlich wurde das Loggen der Anfragen, und der Healthcheck in Middlewares ausgelagert. Es ist auch zu erkennen, dass sich der Code für die Registrierung der FaaS-Funktion von 25 Zeilen auf 15 Zeilen reduziert hat.

```
exports.handler = new Wrapper()
  .withCloudProvider('AWS')
  .withMethod(
    new Method('GET', executeSeminarRequest, [
      readConfiguredRoles,
      prepareValues,
      readConfig,
      validateRequestParameters,
      tryGetCustomer,
      checkPermissions,
    ]),
  )
  .withMiddleware([logRequest, checkHealth])
  .withErrorMiddleware([handleError])
  .build();
```

Abbildung 12: Lambda mit Framework

```
exports.handler = async (event, context) => {
  logLambdaRequestEvent(event);

  const healthStatus = checkHealth(event, context);
  if (healthStatus) {
    return healthStatus;
  }

  const redisOptions = {
    host: event.stageVariables.redisHost,
    port: event.stageVariables.redisPort,
  };

  return Promise.resolve()
    .then(() => readConfiguredRoles(redisOptions, event.requestContext.stage))
    .then(() => prepareValues(event))
    .then((values) => readConfig(values))
    .then((values) => validateRequestParameters(values))
    .then((values) => tryGetCustomer(values))
    .then((values) => checkPermissions(values))
    .then((values) => executeSeminarRequest(values))
    .then(responses.ok)
    .then(appResolve())
    .catch(appReject());
};
```

Abbildung 13: Lambda ohne Framework

7.2 Unabhängigkeit vom Cloudanbieter

Um zu überprüfen, ob eine mit dem Framework entwickelte Anwendung unabhängig vom Cloudanbieter betrieben werden kann, wurde die Beispielanwendung verwendet. Diese konnte erfolgreich auf Azure Functions und AWS Lambda ausgeführt werden. Zu Google Cloud Functions bestand kein Zugang, weshalb die Anwendung nur lokal getestet werden konnte.

7.3 Routing nach HTTP-Anfragemethoden

In der Beispielanwendung wurden ein GET-Endpunkt und ein POST-Endpunkt mit jeweils eigener Middleware, sowie gemeinsamer Middleware verbaut. Beim Test wurden Anfragen mit den verschiedenen HTTP-Methoden an die Anwendung gesendet und von der Anwendung basierend auf der HTTP-Methode der Anfrage die korrekten Antworten zurückgegeben.

8 Fazit

Die Umsetzung konnte nicht ausreichend getestet und auf die Erfüllung der Anforderungen geprüft werden. Stattdessen wurde das im Rahmen der Projektarbeit entwickelte Framework mit einer Beispielanwendung auf zwei verschiedenen Cloudanbietern getestet. Hier konnte das Framework zeigen, dass es dem Entwickler ermöglicht Code mittels des Middleware-Patterns auszulagern, es sich auf verschiedenen Cloudanbietern ausführen lässt, und abhängig von der HTTP-Methode der Anfrage unterschiedliche Middlewares und Endpunkte aufrufen kann.

Es hat sich als schwieriger als erwartet herausgestellt, das Framework in das Kundenprojekt zu integrieren. Das korrekte Umschreiben der Lambdas, sodass sie das Framework verwenden, ist dabei herausfordernd. Das Framework wird nicht im Kundenprojekt verwendet, da der Aufwand, die Lambdas umzuschreiben nicht vertretbar ist.

Dennoch kann der Einsatz des Frameworks bei späteren Projekten, die auf Functions as a Service aufbauen, in Erwägung gezogen werden, da insbesondere die Cloudunabhängigkeit und das Middleware-Pattern von Vorteil sein können.

Literaturverzeichnis

[Ama24] Amazon Web Services, Inc.: „AWS Lambda Data Processing - Datenverarbeitungsdienste“.
https://aws.amazon.com/de/lambda/
Abruf: 2024.03.19

[Cas24] Casco, Emanuel: „GitHub - emanuelcasco/azure-middleware: Node.js middleware engine for Azure Functions ³“.
https://github.com/emanuelcasco/azure-middleware
Abruf: 2024.03.14

[Doc24] Documentation | NestJS - A progressive Node.js framework: „Documentation | NestJS - A progressive Node.js framework“.
https://docs.nestjs.com/faq/serverless
Abruf: 2024.03.14

[Exp24] Express: „Using Express middleware“.
https://expressjs.com/en/guide/using-middleware.html
Abruf: 2024.03.14

[Fpt19] Fptava: „modofun“.
https://modofun.js.org/
Abruf: 2024.03.14

[GK24] Gazis, Alexandros; Katsiri, Eleftheria: „Middleware 101“.
https://dl.acm.org/doi/fullHtml/10.1145/3546958
Abruf: 2024.03.19

[Goo24] Google Cloud: „Cloud Functions | Google Cloud“.
https://cloud.google.com/functions/
Abruf: 2024.03.19

[Hua23] Huawei Technologies Co., L.: „Cloud Computing Technology“, 1. Auflage, Springer Nature Singapore, Singapore, 2023

[Mer24] Merlicco, Yves: „GitHub - yvele/azure-function-express: Allows Express.js usage with Azure Functions“.
https://github.com/yvele/azure-function-express
Abruf: 2024.03.14

[Mic24] Microsoft Azure: „Azure Functions – serverlose Funktionen im Computing | Microsoft Azure“.
https://azure.microsoft.com/de-de/products/functions/
Abruf: 2024.03.19

[Red24] Red Hat, Inc.: „Was ist Middleware? Definition - Arten - Einsatz | Red Hat DE“.
https://www.redhat.com/de/topics/middleware/what-is-middleware
Abruf: 2024.02.22

[Tur24] Turing: „A Complete Guide on How to Build Middleware For Node.js“.
https://www.turing.com/kb/building-middleware-for-node-js
Abruf: 2024.03.19

[Wik24] Wikipedia: „Middleware“, 2024.
https://de.wikipedia.org/w/index.php?title=Middleware&oldid=241034099
Abruf: 2024.03.19