Plattform zur Integration einer externen Recommendation Engine in eine Gamified Learning Platform auf Amazon Web Services

Projektarbeit 1 (T3_1000)

im Rahmen der Prüfung zum Bachelor of Science (B.Sc.)
des Studienganges **Informatik**
an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

Dominik Ochs

Sperrvermerk

Abgabedatum:	04. Oktober 2021
Matrikelnummer, Kurs:	2847475, TINF20B2
Ausbildungsfirma:	SAP SE Dietmar-Hopp-Allee 16 69190 Walldorf, Deutschland
Betreuer der Ausbildungsfirma:	Niraj Singh
Gutachter der Dualen Hochschule:	Heinrich Braun

Sperrvermerk

Die nachfolgende Arbeit enthält vertrauliche Daten der:

SAP SE
Dietmar-Hopp-Allee 16
69190 Walldorf, Deutschland

Der Inhalt dieser Arbeit darf weder als Ganzes noch in Auszügen Personen außerhalb des Prüfungsprozesses und des Evaluationsverfahrens zugänglich gemacht werden, sofern keine anderslautende Genehmigung vom Dualen Partner vorliegt.

Hinweis zu Warenzeichen und Markennamen

Diese Arbeit enthält Nennungen von Unternehmensmarken, Produkten und Dienstleistungen, sowohl der SAP SE als auch anderer Firmen. Diese Nennungen stellen keine Markenzeichenbenutzung im geschäftlichen Verkehr dar und dienen lediglich einem wissenschaftlichen Zweck. Aus Gründen der besseren Lesbarkeit wird somit auf die Kennzeichnung dieser Marken mit den entsprechenden Markensymbolen verzichtet.

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich meine Projektarbeit 1 (T3_1000) mit dem Thema:

> *Plattform zur Integration einer externen Recommendation Engine in eine Gamified Learning Platform auf Amazon Web Services*

gemäß § 5 der „Studien- und Prüfungsordnung DHBW Technik“ vom 29. September 2017 selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Karlsruhe, den 17. Dezember 2020

Gez. Dominik Ochs
Ochs, Dominik

Gleichstellungshinweis

Zur besseren Lesbarkeit wird auf geschlechtsspezifische Doppelnennungen verzichtet.

Abstract

- English -

This paper is about the integration of a *Recommendation Engine (RE)* into a *Gamified Learning Platform (GLP)* using the hyperscaler *Amazon Web Services (AWS)* and as such assesses several AWS technologies in order to find the best ones.

To that end relational databases were compared to graph databases which revealed that graph databases are technically superior. Additionally, several technologies for secure data transmission such as HTTPS, gRPC, and AWS Virtual Private Cloud are considered. Furthermore, requirements for a modern, scalable microservice are analyzed. On that foundation an architecture including necessary technologies like reverse proxies, load balancer and databases is designed. At last, the challenges regarding the implementation of such a platform - integrating an RE into a GLP - are scrutinized.

Abstract

- Deutsch -

Diese Arbeit behandelt das Integrieren einer *Recommendation Engine (RE)* in eine *Gamified Learning Platform (GLP)* auf dem Hyperscaler *Amazon Web Services (AWS)* und untersucht im Zuge dessen einige Technologien von AWS, um die Besten für das Vorhaben zu wählen.
Dabei werden zum Beispiel relationale Datenbanken Graphdatenbanken gegenübergestellt, wobei letztere technisch klar überlegen sind. Auch werden verschiedene Technologien wie HTTPS, gRPC und AWS Virtual Private Cloud zur sicheren Datenübermittlung betrachtet. Weiter werden Anforderungen für einen modernen, skalierbaren Microservice analysiert, worauf dann eine Architektur mit nötigen Technologien wie Reverse-Proxys, Load-Balancer und Datenbanken aufgebaut wird. Schließlich wird ein Blick auf die Herausforderung bei der Implementierung einer solchen Plattform zur Integration der RE in die GLP untersucht.

Inhaltsverzeichnis

Abkürzungsverzeichnis **VI**

Abbildungsverzeichnis **VIII**

Tabellenverzeichnis **IX**

Quellcodeverzeichnis **X**

1. Motivation **1**
1.1. SAP Digital Heroes 1
1.2. Danos Recommendation Engine 2
1.3. Integration 3
1.4. Zielsetzung, Inhalt und Vorgehensweise 4

2. Grundlagentechnologien **5**
2.1. Go 5
2.2. Graphdatenbanken 11
2.3. Amazon Web Services 17

3. Planungsphase **19**
3.1. Anforderungen 19
3.2. Architektur 21
3.3. Voraussetzungen 27

4. Implementierung **30**
4.1. Exponierte API und Cache 30
4.2. REST API 34
4.3. Datenbankadapter 38

5. Fazit **44**
5.1. Zusammenfassung 44
5.2. Kritische Reflexion 45
5.3. Kostenbetrachtung 46

Literaturverzeichnis **XI**

A. Anhang **XVI**
A.1. Evaluation of Expenditures (English Version of *5.3 Kostenbetrachtung*) . XVI
A.2. Dokumentation XVII

Abkürzungsverzeichnis

AI	Artificial Intelligence
API	Application Programming Interface
AWS	Amazon Web Services
CA	Certificate Authority
CIO	Chief Information Officer
CRUD	Create, Read, Update, Delete
DANOS	Dynamic Anthropomorphic Network of Objects Simulator
DB	Database
DH	Digital Heroes
DHP	Digital Heroes Platform
DHGLP	Digital Heroes Gamified Learning Platform
DSGVO	Datenschutzgrundverordnung
EC2	Amazon Elastic Compute Cloud
GDB	Graph Database
GLP	Gamified Learning Platform
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IaaS	Infrastructure as a Service
JSON	JavaScript Object Notation
NAS	Network Attached Storage
NoSQL	Not only Structured Query Language
RE	Recommendation Engine
REST	Representational State Transfer

RPC	Remote Procedure Call
SQL	Structured Query Language
SSH	Secure Shell
SSL	Secure Sockets Layer
TLS	Transport Layer Security
VM	Virtual Machine
VPC	Virtual Private Cloud

Abbildungsverzeichnis

2.1. Frage nach der Nutzung von Programmiersprachen, Skriptsprachen und Markup-Sprachen unter professionellen Entwicklern. (Abb. entnommen aus [20]) 6
2.2. Prozentualer Anteil ausgewählter Sprachen an Pull-Requests auf Github zum jeweiligen Zeitpunkt. (Abb. entnommen aus [23]) 7
2.3. Prozentuale Verteilung der Popularität (Ranking-Punkte) auf die verschiedenen Datenbankmodelle. (Abb. entnommen aus [32]) 13
2.4. Normalisierter Anstieg der Popularität (Ranking-Punkte) der verschiedenen Datenbankmodelle in den letzten sieben Jahren. (Abb. entnommen aus [32]) 14
2.5. Marketshare verschiedener Infrastructure as a Service (IaaS)-Anbieter stand Q2 2020. (Abb. entnommen aus [39]) 17

3.1. Äußere Architektur (Blackbox) von *Proteus*. Hier können auch noch weitere parallele *Proteus*-Instanzen eingefügt werden. (Abb. entnommen aus eigener Arbeit) 25
3.2. Innere Architektur (Whitebox) von *Proteus*. (Abb. entnommen aus eigener Arbeit) 27

4.1. Implementierungshierarchie von *Cache* und dessen *Interfaces* (gekürzt). (Abb. entnommen aus eigener Arbeit) 31
4.2. Sequenzdiagramm zum Ablauf eine Authentifizierung und Autorisierung (vereinfacht). (Abb. entnommen aus eigener Arbeit.) 37

Tabellenverzeichnis

5.1. Formeln zu Berechnung der monatlichen Kosten; mit n der Anzahl an DHP-Instanzen. 47

A.1. Montly Expenditure Formulas; n being the number of DHP instances. . . XVII

Quellcodeverzeichnis

2.1 Nebenläufigkeit durch das *go* Schlüsselwort. 8
2.2 Go-Struktur *Manager* mit den Gleitkommazahl-Feldern *salary* und *bonus*. 9
2.3 Go Receiver-Methode für die Manager-Struktur. 9
2.4 Definition einer Struktur *Server* und ihrer Receiver-Methode *AnualExpenditure*. 9
2.5 Definition eines Interfaces *Expenditure* mit der Methode *AnualExpenditure* mit leerer Parameterliste und einer Gleitkommazahl als Rückgabetyp. . . 10
2.6 *main*-Funktion iteriert über ein *Slice* (vgl. Array) von *Expenditure*s - das zuvor mit einem Manager und einem Server befüllt wurde - um die Ausgaben aufzusummieren. 11

4.1 Existenzüberprüfung eines bestimmten Knoten nach ID. 32
4.2 Initialisierung des *Proteus*-Services. 34
4.3 *Proteus Permissions*-Datei-Template mit API-Keys und Berechtigungen. 36
4.4 Funktion die bei POST von Heroes und Missionen aufgerufen wird. Label bestimmt den Typ. 38
4.5 Interne Darstellung eines Knoten des Packages *Graph*. 39
4.6 Antwort-JSON von Neptune (Ausschnitt). 39
4.7 Hinzufügen eines Knoten über *Query*-Datenbankadapter (angepasst) . . . 42
4.8 Erstellen der Eigenschaftsliste von Knoten und Kanten für eine Gremlin-Abfrage 42

1. Motivation

Im Folgenden wird die Motivation für diese Arbeit, sowie die Relevanz dieser Arbeit in dem Kontext einer *Gamified Learning Platform (GLP)* erläutert. Dafür erfolgt zunächst eine Einführung in die beiden Technologien der *Digital Heroes Gamified Learning Platform (DHGLP)* und der *Danos Recommendation Engine (RE)*, und warum es die RE in die GLP zu integrieren gilt. Daran anschließen wird sich die genaue Zielsetzung, der Inhalt und die Vorgehensweise dieser Arbeit.

1.1. SAP Digital Heroes

Die *Digital Heroes (DH)-Initiative* ist eine firmeninterne Plattform der SAP SE zum Teilen von Wissen, insbesondere bezüglich unterschiedlichen internen SAP Tools [1]. So haben der ehemalige Chief Information Officer (CIO) Thomas Saueressig [1], sowie der neue CIO Florian Roth [2] beide zugegeben, dass es in der SAP durchaus ein Tool-Chaos geben könnte, wobei es manchmal schwierig sei, die richtigen Werkzeuge zum richtigen Zeitpunkt für den richtigen Einsatz zu finden, zu verstehen und dann auch erfolgreich zu benutzen [1, 3].
Dieses Problem sucht die DH-Initiative zu lösen und hat dafür eine *Gamified Learning Platform (GLP)* geschaffen, die durch *Gamification*-Aspekte versucht Wissen locker, leicht, und verständlich über die *Heroes* - die tatsächlich in ihrem täglichen Arbeitsleben damit umgehen - zu vermitteln [1].
Dies geschieht über *Missionen*, die von den Mitgliedern (Heroes) der DHGLP erstellt werden [1]. Ein Hero kann dabei eine Mission erledigen, indem er die Aufgaben dieser Mission bewältigt. Für das Beenden einer Mission erhält er Punkte und wird in Leaderboards angezeigt.
Die DHGLP benutzen und Hero werden kann dabei jeder SAP Mitarbeiter um Wissen zu teilen und zu erlangen [1].
Hier geht es zur Digital Heroes Platform (https://digital-heroes.int.sap/#/hero/home).

Das Backend der Digital Heroes Gamified Learning Platform (DHGLP) ist in Java geschrieben und läuft auf einer *SAP Neo-Cloud*-Instanz mit einer *SAP HANA Database (DB)* und dem *SAP Document Service* zur persistenten Datenhaltung [4]. Das DH Backend bietet für das Vue.js Frontend ein *Representational State Transfer (REST) Application Programming Interface (API)* zur Kommunikation per *Hypertext Transfer Protocol Secure (HTTPS)* [4].

1.2. Danos Recommendation Engine

Die

> "[...] DANOS (Dynamic Anthropomorphic Network of Objects Simulator)." [5, p. 1]

(im Folgenden *Danos*) *Recommendation Engine (RE)* ist eine Artificial Intelligence (AI) Technologie zum Vorschlagen von Objekten, die einen Nutzen für den User haben könnten [5, 6, 7]. Damit ist die Danos-RE general-purpose und kann somit für alle möglichen Systeme eingesetzt werden, um den (End-)Nutzern Vorschläge zum besseren und nutzerfreundlicheren Verwenden des Systems zu liefern [5, 7]. Das können zum Beispiel Vorschläge zu Musik auf Musik-Streaming-Seiten, Artikel in Zeitungen, oder Produkte auf Webshops sein - eben alles, was Entscheidungsprozesse des Nutzers beinhaltet [6]. Hierfür arbeitet Danos mit sogenannten *User-Specifics* (z. B. Vorzügen und Charaktereigenschaften) für die Nutzer, und *Object-Specifics* (z. B. Gegenstandscharakteristiken) für die Objekte, um so ein Netz aus Nutzern und Objekten aufzubauen [7]. Zwischen allen Objekten (Knoten) des daraus entstehenden Graphs werden über verschiedene Ähnlichkeitsmetriken Verbindungen (Kanten) - sogenannte *Freundschaften* - aufgebaut [7]. Über diese Freundschaften werden dann die Vorschläge (Recommendations) errechnet [5, 7].
Während das Danos-Kernsystem mit aller Art von Nutzern und Objekten arbeiten kann, müssen für die Danos-RE applikationsspezifische Anpassungen vorgenommen werden [5].

Danos wurde im Frühjahr 2020 als Accelerator-Projekt bei SAP gestartet [8], was bedeutet, dass das SAP-externe Entwicklerteam von Danos - Panagiotis Germanakos und Daniel Defiebre [8] - die Chance hat einen Prototypen ihrer RE mit SAP-Ressourcen und auf der SAP-Plattform kostenlos zu entwickeln und zu testen - die Entwicklung also durch

SAP gesponsored wird [9]. Um diesen Prototypen zu entwickeln und zu testen, wird die DHGLP verwendet [8].

Danos ist in der Programmiersprache Go geschrieben und plattformunabhängig einsetzbar. Der Quellcode von Danos liegt **nicht** bei der SAP SE und ist für sie (und mich als SAP-Angestellter) nicht direkt zugänglich.

1.3. Integration

Fortan wird *Digital Heroes Platform (DHP)* für *Digital Heroes Gamified Learning Platform (DHGLP)* verwendet und bezeichnet sowohl die vom Nutzer sichtbare Plattform selbst, als auch deren technische Implementierung auf der Neo-Cloud. Mit DHP wird also der ganze DH-Technologie-Stack bezeichnet, der in Abschnitt 1.1 vorgestellt wurde.

Wie in Abschnitt 1.2 bereits erklärt, soll die DHP verwendet werden, um Danos weiterzuentwickeln. Die DHP bietet natürlich den optimalen Use-Case für Danos, welches hier in Zukunft den Heroes Missionen vorschlagen soll, welche diesen gefallen könnten. Von dieser Kooperation haben beide Parteien etwas - so kann Danos in einem Real-Life-Szenario erprobt werden, während die DHP herausragende Vorschläge für ihre Nutzer anbieten kann. Wenn die Integration und der Prototyp Erfolg haben, öffnet das viele Möglichkeiten für die SAP SE, sowie das Danos-Team die RE in weiteren Plattformen und Systemen zu integrieren, was schlussendlich einen Nutzen für die SAP Kunden haben kann und die SAP von ihrer Konkurrenz in diesem Bereich abheben könnte. Der kommerzielle Nutzen liegt auf der Hand und würde dazu beitragen die SAP auch in Zukunft weithin als Wachstumsunternehmen zu qualifizieren.

Zur Integration von Danos in die DHP wird eine stabile, solide, skalierbare Infrastruktur benötigt, auf der die Danos-RE laufen kann, sodass in Zukunft auch weitere Integrationen leicht und mit möglichst wenig Aufwand durchgeführt werden können. Hierzu soll eine Integration über den Hyperscaler Amazon Web Services (AWS) - die die Anforderungen nach eigenen Aussagen erfüllen [10] - erfolgen.

1.4. Zielsetzung, Inhalt und Vorgehensweise

Ziel dieser Arbeit ist es die Danos-RE in die DHP zu integrieren. Dazu soll die Software-Infrastruktur, die dann auf der AWS-Hardware-Infrastruktur läuft, aufgebaut werden, mit der die Danos-RE mit der DHP kommunizieren kann. Diese Software-Infrastruktur muss dazu bestehen aus:

- einer REST API zur Kommunikation zwischen Danos und der DHP,
- einem Datenbankadapter für die *Graph Database (GDB)*, die den von Danos errechneten Graphen speichern muss,
- mehreren Parsern, die zwischen den verschiedenen Datenformaten (JavaScript Object Notation (JSON), GDB-Strukturen, Danos-Strukturen) übersetzen müssen,
- einer Go API welche von Danos konsumiert wird, um mit der DB und der DHP zu kommunizieren.

Außerdem wird es Aufgabe sein, die AWS-Hardware-Infrastruktur aufzusetzen und zu Administrieren. Hierfür wird eine general-purpose Serverinstanz zum Beherbergen der Danos-Engine und dieser Arbeit auf AWS sowie eine GDB auf AWS benötigt. Auch wird ein strenges Sicherheitskonzept aufgrund der vielen sensiblen Mitarbeiterdaten (vgl. Präferenzen, Charaktereigenschaften) obligatorisch, um die Datenschutzgrundverordnung (DSGVO) einzuhalten.
Um diesen Anforderungen gerecht werden zu können, müssen die notwendigen und ausgewählten Technologien in einem Grundlagenkapitel untersucht werden, daraufhin müssen die Anforderungen genau analysiert und das Projekt geplant werden, um das Ausmaß der Implementation feststellen zu können. Abschließend eine Zusammenfassung und kritische Bewertung des Projektes durchgeführt.

2. Grundlagentechnologien

Im Folgenden werden die essentiellen Technologien untersucht, die für die Umsetzung des Projektes unerlässlich sind. Dazu wird zuerst auf die Programmiersprache Go, deren Relevanz und einige Eigenschaften zusammen mit wichtigen Sprachkonstrukten eingegangen. Außerdem müssen GDB zusammen mit der Query-Language *Gremlin* betrachtet werden, sowie ein kurzer Blick auf die verwendeten AWS-Technologien geworfen werden.

2.1. Go

Go - oder auch Golang (wobei der offizielle Name Go lautet [11]) - ist eine statisch-typisierte, kompilierte, nebenläufige und automatisch-speicher-bereinigte Open-Source-Programmiersprache [12, 13], die hauptsächlich von Google entwickelt wird [14]. Die Sprache wurde zum ersten Mal im November 2009 vorgestellt [14, 15] und die erste stabile Version (Go 1) wurde im März 2012 herausgebracht [16]. Go wurde von Robert Griesemer, Rob Pike und Ken Thompson designed [15], welche folgende Designziele festlegten: Die Sprache solle

- die Effizienz einer statisch-typisierten, kompilierten Sprache, aber die Einfachheit einer dynamischen Sprache haben,[17]
- sicher sein: Typsicher und sichere Speicherzugriffe, [17]
- gute Unterstützung für Nebenläufigkeit und Kommunikation haben, [17]
- effiziente, latenz-freie, automatische Speicherbereinigung haben, [17]
- und super schnell kompilieren. [17]

Hauptgrund für den Entwicklungbeginn der Sprache von Googles Seite aus, war es eine Programmiersprache zu schaffen, die Lücken in bestehenden Sprachen füllt und deren gute Aspekte in die neue Sprache Go übernimmt [15].

2.1.1. Sprachrelevanz

Go hat sich seit seiner Erstveröffentlichung im Jahr 2012 [16] zu einer äußerst beliebten Programmiersprache, vor allem für Backends, entwickelt [18]. Laut Robert Pike (einem der Designer der Sprache; vgl. 2.1) sei dies vor allem der Fall, da die Programmiersprachen, die dafür sonst verwendet werden (Python, Java, C++) zu einer Zeit und in einem Software-Umfeld entwickelt wurden, welches dem heutigen kaum mehr gleicht [18].

Weiter kann die Beliebtheit und der Nutzen der Sprache durch eine Vielzahl verschiedener Metriken überprüft werden. Eine dieser Metriken ist dabei der jährliche Stack Overflow Developer Survey. Stack Overflow ist eine Question and Answer (Q&A) Website über Programmierfragen. Hier können im Forum Hobby- und professionelle Entwickler Programmierthemen diskutieren [19]. Der Survey stellt den Nutzern verschieden Fragen, um unter anderem die Benutzung verschiedener Programmiersprachen zu ermitteln. Für die Umfrage 2020 wurden circa 80 000 Mitglieder befragt [20]. Eine der Grafiken die daraus hervorgeht ist die Folgende:

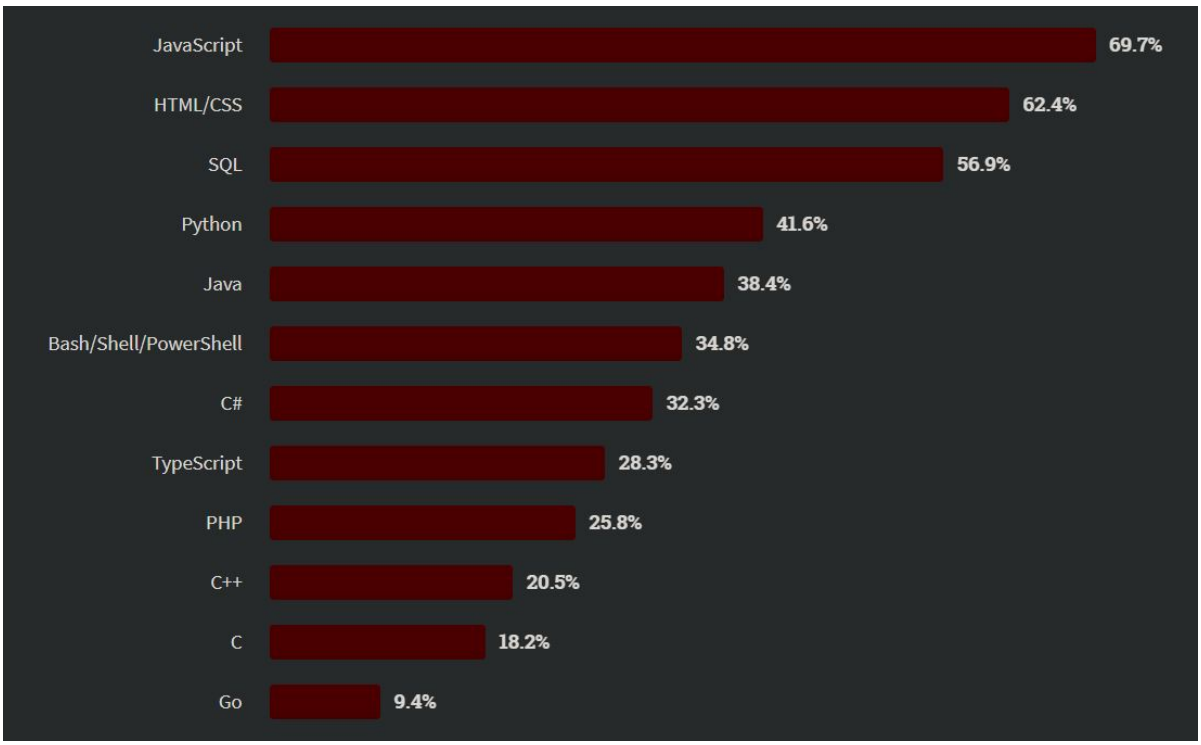

Abbildung 2.1.: Frage nach der Nutzung von Programmiersprachen, Skriptsprachen und Markup-Sprachen unter professionellen Entwicklern.
(Abb. entnommen aus [20])

In Abbildung 2.1 sind die meistbenutzen Programmier-, Skript- und Markup-Sprachen unter professionellen Entwicklern auf der Plattform Stack Overflow aufgeführt. Hier belegt Go den 12. Platz und wird damit von 9,4% der professionellen Entwicklern auf Stack Overflow genutzt. Lässt man die Skript- und Markup-Sprachen außer Acht, so belegt Go sogar Platz 9.

Eine weitere wichtige Metrik kann von der populären Open-Source-Code-Hosting- bzw. Git-Versionskontroll-Hosting-Plattform *Github* bezogen werden. Die Relevanz und Bedeutsamkeit dieser Plattform lässt sich am besten in Zahlen ausdrücken: So sind rund 56 Millionen Entwickler auf der Plattform und 72% der Fortune 50 Companies (inklusive der SAP SE) benutzen Github [21]. Auf Github wird die aktuelle (!) Relevanz einer Sprache durch die Anzahl an sogenannten *Pull-Requests* in einer bestimmten Sprache gemessen. Eine Pull-Request ist dabei eine Art Update-Anfrage auf ein gehostetes Git(hub)-Projekt [22].

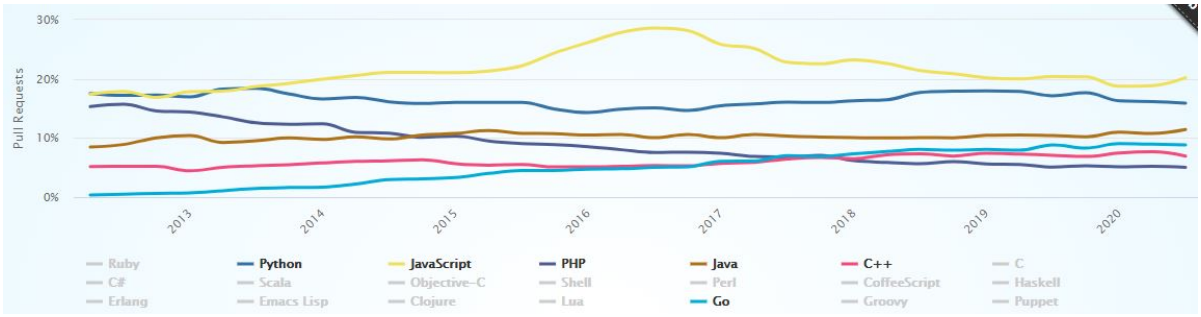

Abbildung 2.2.: Prozentualer Anteil ausgewählter Sprachen an Pull-Requests auf Github zum jeweiligen Zeitpunkt.
(Abb. entnommen aus [23])

In Abbildung 2.2 lässt sich der prozentuale Anteil an Pull-Requests (PR) auf Github für ausgewählte Sprachen und zu bestimmten Zeitpunkten erkennen. Hierbei bildet JavaScript (hellgelb) mit 20,17% der PR die alleinige Spitze im Dezember 2020. Go (hellblau) hat 8,82% der Pull-Requests und es ist über die Jahre ein stetiger Anstieg mit nach wie vor steigender Tendenz zu erkennen. Auch lässt sich sehen, dass andere Sprachen wie Java (braun) und C++ (pink) im Gegensatz zu Go in ihrem Anteil gesättigt scheinen, oder sogar an Relevanz verlieren, wie z. B. PHP (violett).

2.1.2. Sprachliche Konstrukte

In diesem Abschnitt werden nun einige der Sprachkonstrukte und Features vorgestellt, die Go von anderen Sprachen (wie C, Java, etc.) abhebt. Zum einen wäre dafür die besonders einfache *Concurrency* (Nebenläufigkeit) von Go zu nennen. Um eine weitere Methode/Funktion (pseudo-)parallel zu einer anderen zu starten, benötigt es lediglich das Schlüsselwort *go* vor der gewöhnlichen Methoden-/Funktionsinvokation [24, S. 87]. Dies ist nochmals in dem folgenden Beispiel dargestellt. Hier fällt auch der sehr minimale Syntax von Go auf.

```
1  package main
2
3  func inParallel() {
4      // ... this runs in parallel to the code in main
5  }
6
7  func main() {
8      // use 'go' to invoke inParallel as a 'Goroutine'
9      go inParallel()
10
11     // infinite loop
12     for true {
13         // ... this runs in parallel to the code in inParallel
14     }
15 }
```

Code-Ausschnitt 2.1: Nebenläufigkeit durch das *go* Schlüsselwort.

Im Weiteren will ich die *Objektorientierung* in Go vorstellen, die nicht wie in anderen Sprachen (Java, C++, C#) auf Klassen basiert, sondern versucht durch ein orthogonales Modell Methoden und Datenstrukturen beliebig und optional zu verbinden [24, S. 55–59, 25]. Im nachstehenden Code-Ausschnitt ist eine Go Struktur dargestellt, die ähnlich wie Strukturen in C funktionieren und auf deren Felder (Primitives, Structures, Intefaces) mit der für C und Java üblichen Punkt-Notation zugegriffen werden kann [24, S. 55–59].

```
1 type Manager struct {
2     salary float64
3     bonus float64
4 }
```

Code-Ausschnitt 2.2: Go-Struktur *Manager* mit den Gleitkommazahl-Feldern *salary* und *bonus*.

Wollen wir nun eine Methode an diese Struktur anheften, die dann ebenfalls mit der Punkt-Notation aufgerufen werden kann, so geschieht das folgendermaßen [24, S. 55–59]:

```
1 func (m Manager) AnualExpenditure() float64 {
2     return m.salary + m.bonus
3 }
```

Code-Ausschnitt 2.3: Go Receiver-Methode für die Manager-Struktur.

Vor dem Methodennamen wird nun die Struktur mit einem Alias (hier 'm') aufgeführt, um die Zugehörigkeit der Methode zu der Struktur auszudrücken. Auf den Alias m kann innerhalb der Methode zugegriffen werden, um weitere Methoden der Struktur aufzurufen oder Feldzugriffe zu ermöglichen [24, S. 55–59]. Der *Receiver* ist also eine Art spezieller Parameter und ist stets das Argument, der Struktur, auf der die Receiver-Methode aufgerufen wird [24, S. 55–59]. Die definierte Methode errechnet dabei die jährlichen Ausgaben, die für diesen Manager notwendig werden, indem das Jahresgehalt und ein eventueller Bonus addiert werden.
Dieser Prozess lässt sich auch für eine andere Struktur, die ebenfalls Ausgaben verursacht wiederholen:

```
1 type Server struct {
2     hourlyCost float64
3     uptimeInHours float64
4 }
5
6 func (s Server) AnualExpenditure() float64 {
7     return s.hourlyCost * s.uptimeInHours
8 }
```

Code-Ausschnitt 2.4: Definition einer Struktur *Server* und ihrer Receiver-Methode *AnualExpenditure*.

Es wird nötig, dass der *Server* ebenfalls eine eigene Receiver-Methode *AnualExpenditure* bekommt, da die Kalkulation anders abläuft, und der Server andere Felder hat. Hier werden die Kosten über das Produkt der stündlichen Kosten für den Server und der jährlichen Laufzeit in Stunden(bruchteilen) berechnet.
Was nun auffällt, ist, dass beide Strukturen jeweils eine Methode *AnualExpenditure* haben, deren Signatur - bis auf den Receiver - gleich ist. In anderen, objektorientierten Programmiersprachen, würde es also Nahe liegen eine Vererbungshierachie über Superklassen oder Interfaces zu implementieren. Und genau über letzteres - Interfaces - können wir auch in Go eine *is-a-Relation* realisieren [24, S. 55–59]. Zunächst wird das Interface definiert:

```
1 type Expenditure interface {
2     AnualExpenditure() float64
3 }
```

Code-Ausschnitt 2.5: Definition eines Interfaces *Expenditure* mit der Methode *AnualExpenditure* mit leerer Parameterliste und einer Gleitkommazahl als Rückgabetyp.

Das definierte Interface enthält nur die Signatur der AnualExpenditure-Methode und in Go - das ist übrigens auch mit 'orthogonalem Modell' gemeint - implementiert eine Struktur **implizit** ein Interface genau dann, wenn die Struktur Receiver-Methoden für jede Signatur des Interfaces definiert und implementiert hat [24, S. 55–59]. Das bedeutet, dass sowohl *Manager* als auch *Server* beides *Expenditure*s sind, weil sie alle Methoden des Interfaces per Receiver-Methode implementieren. Diese is-a-Beziehung wird in folgendem Beispiel deutlich, in dem sowohl der Manager, als auch der Server in eine Sammlung von Expenditure-Interfaces hineinpassen - dynamische Polymorphie [26]:

```
1  var expenditures = []Expenditure{
2     Manager{salary: 120500, bonus: 10500},
3     Server{hourlyCost: 9.99, uptimeInHours: 1337},
4  }
5
6  func main() {
7     var totalExpenditure float64
8
9     for i := 0; i < len(expenditures); i++ {
10       totalExpenditure += expenditures[i].AnualExpenditure()
11    }
12 }
```

Code-Ausschnitt 2.6: *main*-Funktion iteriert über ein *Slice* (vgl. Array) von *Expenditure*s - das zuvor mit einem Manager und einem Server befüllt wurde - um die Ausgaben aufzusummieren.

2.2. Graphdatenbanken

Graph Databases (GDBs) repräsentieren ihre Daten als Graph-Struktur, also als Sammlung von Knoten (Nodes, Vertices) und Kanten (Edges) zusammen mit Eigenschaften (Properties), wobei Kanten immer als (gerichtete) Verbindung zwischen zwei Knoten dienen [27]. Somit ist es einfach Daten zu verarbeiten, die bereits in diesem Format vorliegen, oder etwa als Graph aufgebaut werden müssen, um sinnvoll weiterverarbeitet werden zu können [27]. Außerdem erlauben GDBs es verschiedene Graph-Kalkulationen durchzuführen, wie zum Beispiel den minimalen Abstand (in Schritten) von zwei Knoten im Graph, mithilfe von verschiedenen Wegfindungsalgorithmen [27]. Das spart es, effiziente Algorithmen hierfür selbst implementieren zu müssen.
GDBs gehören zu den *Not only Structured Query Language (NoSQL)*-Datenbanken [28], d. h. sie können nicht oder nur bedingt über klassisches Structured Query Language (SQL) angesprochen werden. Die häufigsten und beliebtesten Query/Traversal Languages für GDBs sind:

> "[...] GraphQL, AQL, *Gremlin*, SPARQL, and Cyper." [29]

Die beliebtesten GDBs sind Neo4j, gefolgt von MS Azure Cosmos DB, und mit Amazon Neptune auf Platz 7 [29].

2.2.1. Relationale versus Graphdatenbanken

Im Weiteren werden GDBs mit relationalen Datenbanken verglichen. Hierbei haben GDBs die folgenden Vorteile gegenüber relationalen Datenbanken:

Pros Graph Databases (GDBs):

- Bessere Performanz beim Abfragen von stark vernetzten Daten, da *Joins* bei relationalen Datenbanken teuer sind. [30, 31]
- Abfragegeschwindigkeit ist nicht abhängig von der Menge der Daten. [31]
- Für Menschen intuitives und sehr anschauliches Datenmodell und Visualisierung, weil das Graphenmodell unserer Vorstellung von Beziehungen optimal gerecht wird. [30, 31]
- Flexibles und agiles Datenmodell, was Veränderung der in der GDB enthaltenen Datenstrukturen leicht ermöglicht, anders als bei den starren und unflexiblen Tabellen der relationalen Datenbanken. [30, 31]

Cons Graph Databases (GDBs):

- Wenn die Daten nicht stark vernetzt sind, zum Beispiel wenn es nur einen Objekttyp, beziehungsweise nur eine Tabelle bei relationalen Datenbanken gibt, oder keine Joins vorkommen, dann ist der Einsatz von GDBs nicht optimal. [30]
- Für GDBs gib es keine standardisierte Abfragesprache wie SQL für relationale Datenbanken, sondern viele unterschiedliche Sprachen (Gremlin, Cypher, GraphQL etc.). [30]
- Wenige erfahrene Entwickler für GDBs, da die Technologie noch recht jung und nicht so beliebt ist wie relationale Datenbanken. [30]

2.2.2. Popularität von Graphdatenbanken

Obwohl die GDBs technisch gesehen klar besser zu sein scheinen, sind sie noch deutlich weniger populär als relationale Datenbanken, wie das nachstehende Diagramm zeigt:

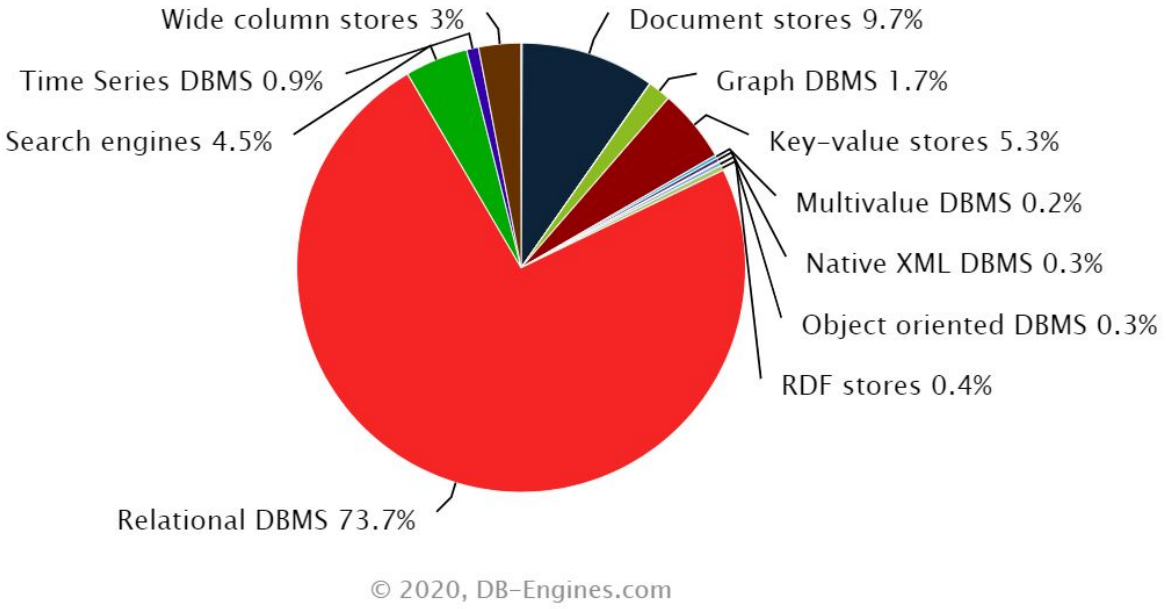

Abbildung 2.3.: Prozentuale Verteilung der Popularität (Ranking-Punkte) auf die verschiedenen Datenbankmodelle.
(Abb. entnommen aus [32])

In obigem Schaubild ist zu erkennen, dass relationale Datenbanken das populärste Datenmodell mit fast drei Vierteln der Ranking-Punkte von *db-engines.com* darstellen. Mit lediglich 9,7% schließen sich die dokumentorientierten DBs an und auf Platz sechs folgen dann schließlich die GDBs mit 1,7%. Es ist also deutlich zu sehen, dass Graphdatenbanken bisher noch eher unpopulär sind, was sich allerdings in Zukunft ändern könnte, betrachtet man die folgende Abbildung:

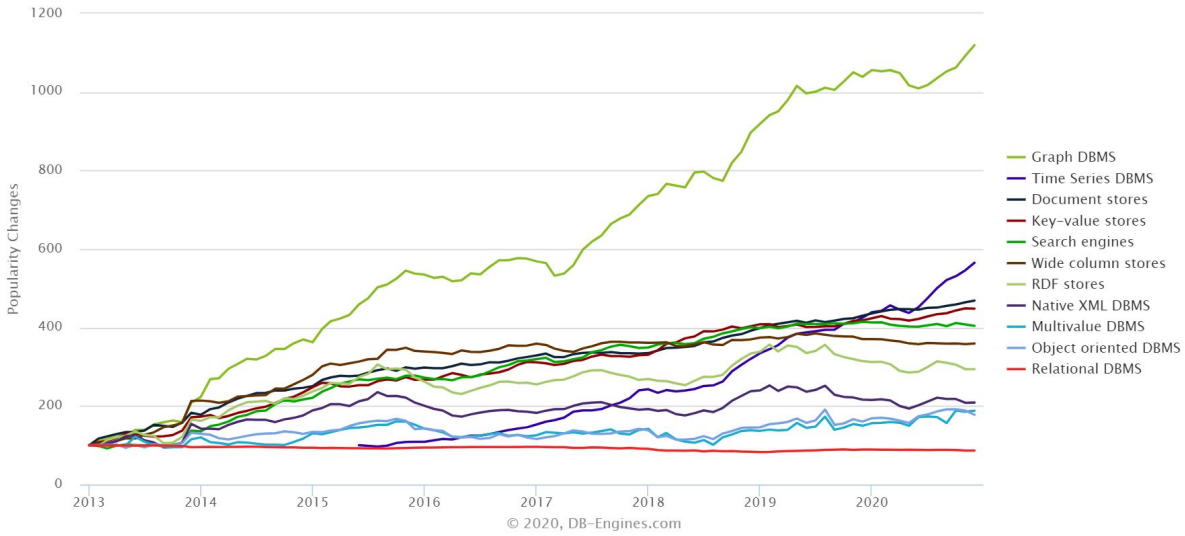

Abbildung 2.4.: Normalisierter Anstieg der Popularität (Ranking-Punkte) der verschiedenen Datenbankmodelle in den letzten sieben Jahren.
(Abb. entnommen aus [32])

In obiger Abbildung stechen die GDBs (hellgrün) klar als das schnellst-wachsende Datenbankmodell hervor, während relationale Datenbanken (orange) keinen, oder kaum Popularitätswachstum zeigen.

2.2.3. Gremlin

Wie in Abschnitt 2.2 bereits erwähnt, zählt *Gremlin* zu einer der weitverbreitetsten Abfragesprachen für GDBs [29], außerdem ist es eine der offiziellen Abfragesprachen der AWS Netpune DB [33]. In diesem Abschnitt werden deswegen die Grundlagen für Gremlin gelegt.
Gremlin ist eine open-source *Graph Traversal Language*, sowie eine *Virtual Machine (VM)* entwickelt von Apache TinkerPop, welche zur Apache Software Foundation gehören [34]. Die VM funktioniert dabei als ein Interpreter der Gremlin-Sprache, um dann graphdatenbankunabhängig das selbe Ergebnis erzielen zu können [35]. Diese Architektur ermöglicht zudem den einfachen Zugriff aud GDBs via verschiedenster Programmiersprachen (für die es viel Unterstützung gibt)[34], wie auch per Hypertext Transfer Protocol (HTTP), womit plain Gremlin an die jeweilige Datenbank übermittelt werden kann, wie im Falle von AWS Neptune [28].

Gremlin funktioniert und selektiert über sogenannte Graphtraversion, was bedeutet, dass eine Gremlin-Query immer mit einer Menge an Elementen (Knoten oder Kanten) beginnt, und dann von dort aus nach und nach die einzelnen Elemente traversiert bzw. durchläuft, um sich dann so nach und nach sich Ergebnis zu nähern [36, S. 29]. So beginnt man mit dem ganzen Graph, der noch alle Elemente (Knoten, Kanten) beinhaltet, indem man diesen einfach mit *g* referenziert [36, S. 29]. Außerdem ist wichtig, dass jeder Knoten und jede Kante in den Graphen ein sogenanntes *Label* hat, welches den Knoten oder die Kante klassifiziert, also das Element zu einer Gruppe mit dem selben Label zuordnen lässt [37]. In relationalen Datenbanken wäre solch ein Label also vergleichbar mit dem Tabellennamen, zu welchem eine Zeile gehört. In Graphdatenbanken sind die Eigenschaften (Properties) für Knoten/Kanten eines Labels allerdings nicht fest, sondern können beliebig sein.

Als Modell für die folgenden Beispiele und Code-Ausschnitte wird angenommen, dass es in der GDB Knoten mit dem Label *Hero* und Knoten mit dem Label *Mission*, sowie Kanten mit dem Label *rates*, die jeweils von einem Hero-Knoten zu verschiedenen Mission-Knoten zeigen, gibt.
In folgendem, ersten Beispiel wird ein neuer Hero-Knoten mit zwei Eigenschaften - Vorname und Nachname - zur DB hinzugefügt:

```
g.addV('Hero').property('FirstName', 'Dominik')
.property('LastName', 'Ochs')
```

Zuerst wird der gesamte Graph mit *g* angesprochen, danach wird zu dieser ganzen Sammlung an Knoten und Kanten, die jetzt zur Verfügung stehen, ein neuer Knoten (**V**ertex) mit dem *Hero*-Label hinzugefügt. Hier könnte der Befehl jetzt schon aufhören - dann wäre ein neuer Hero hinzugefügt, der allerdings überhaupt keine Eigenschaften (außer einer ID) hat [37]. Die Eigenschaften 'FirstName' ist an diesen Hero überhaupt nicht angeheftet (undefiniert) und würde auch keinen uninitialisierten Wert zurück geben, wie man das von einer relationalen Datenbank erwarten würde [37]. Stattdessen liefert der addV-Befehl allerdings den neu-erzeugten Hero-Knoten zurück und man kann diese Referenz nutzen, um nun noch weitere Operationen auf diesem Hero-Knoten auszuführen, wie z. B. Eigenschaften hinzuzufügen [37]. Es gibt ein paar besondere Eigenschaften, wie z. B. die ID eines Knotens oder einer Kante, die keine Anführungszeichen als Eigenschafts-Namen benötigen [37]. Um also einen Hero mit einer custom ID hinzuzufügen, benötigt man folgenden Befehl:

```
g.addV('Hero').property(id, 'my-custom-id')
```

(Hierbei ist allerdings zu beachten, dass der Datentyp der ID von der Datenbank abhängt und nicht immer unbedingt ein String sein muss, wie es bei Neptune der Fall ist [37, 38].)

Um nun über die ID auf einen bestimmten Knoten oder Kante direkt zuzugreifen, lässt sich dieser Befehl verwenden [37]:

```
g.V('my-custom-id')        g.E('my-custom-edge-id')
```

Eine Kante lässt sich folgendermaßen hinzufügen [37]:

```
g.V('hero-id').addE('rates')
.to(V('mission-id')).property('prop','val')
```

Um noch eine reine Abfrage vorzuführen, beschreibt folgender Befehl, wie man alle Heroes abfragen und zurückgeliefert bekommen könnte [37]:

```
g.V().hasLabel('Hero')
```

2.3. Amazon Web Services

Amazon Web Services (AWS) ist ein Hyperscaler der Amazon.com, Inc. [10]. Er ist wohl der größte und einflussreichste *Infrastructure as a Service (IaaS)*-Anbieter, wie das nachfolgende Diagramm veranschaulicht:

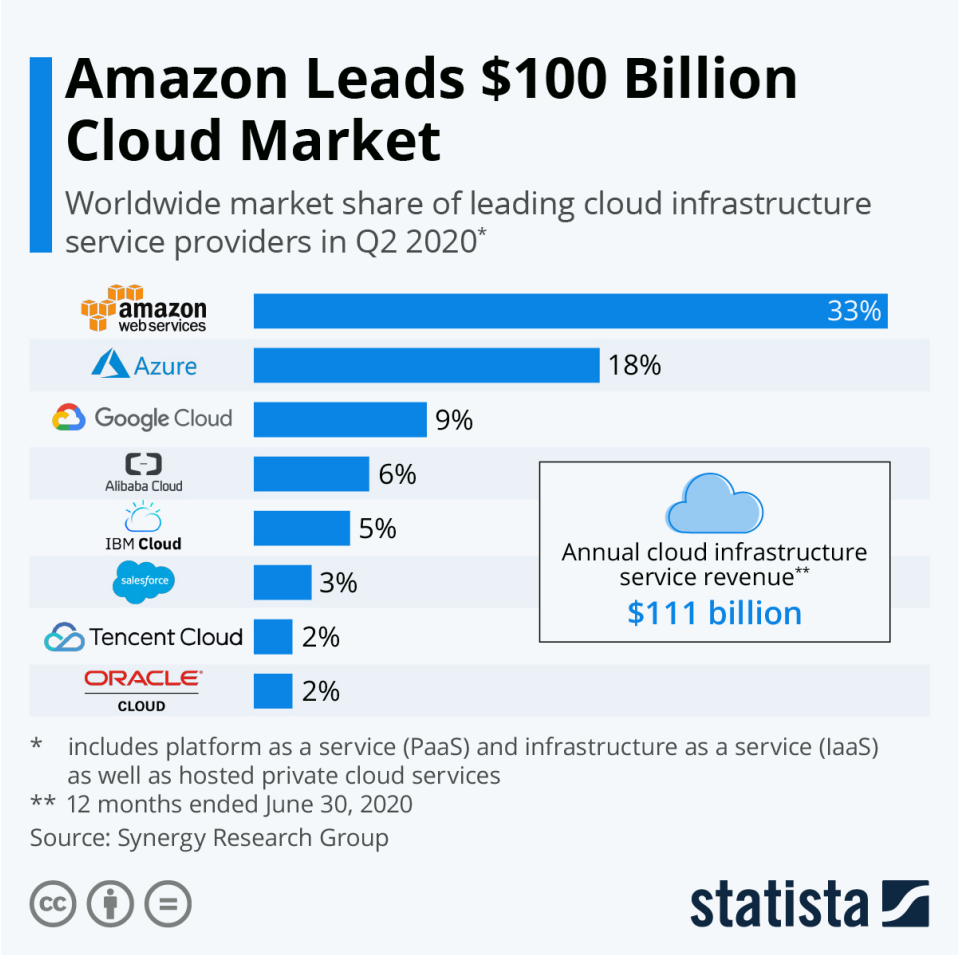

Abbildung 2.5.: Marketshare verschiedener IaaS-Anbieter stand Q2 2020. (Abb. entnommen aus [39])

Für diese Arbeit ist er der Host der Wahl, und die beiden hauptsächlich für diese Arbeit relevanten Technologien - *Amazon Elastic Compute Cloud (EC2)* und die GDB *AWS Neptune* - werden nun weiter beleuchtet.

2.3.1. Amazon Elastic Compute Cloud (EC2)

Amazon Elastic Compute Cloud (EC2) ist ein Teil von Amazon Web Services (AWS), der es ermöglicht virtuelle Computer-Ressourcen bzw. virtuelle Computer zu mieten, worauf dann alle mögliche Software gelaufen lassen werden kann [40]. Amazon vermietet hierbei sogenannte *Container* bzw. *Instanzen* die praktisch eine sehr leicht-gewichtige Virtual Machine (VM) darstellen und vom Kunden komplett eigenständig konfiguriert werden können [40]. Bei der Erstellung kann hierbei zwischen verschiedenen Betriebssystemen wie zum Beispiel Amazon Linux 2, Debian Server, Windows Server, macOS gewählt werden, wofür der Kunde dann einen Secure Shell (SSH)-Zugang mit Administrator-Privilegien bekommt, der genutzt werden kann, um auf den Server zuzugreifen [41].
Amazon wirbt ganz besonders mit der Flexibilität von EC2-Instanzen und erlaubt so Instanzen on-demand starten und stoppen zu können, und nur für die aktivierte Zeit abzurechnen [40]. So reichen die Preise von ca. $0,026 pro Stunde für 1 virtuellen Kern und 2 GiByte Arbeitsspeicher bis ca. $5,4 pro Stunde für 96 virtuelle Kerne mit 384 GiByte Arbeitsspeicher [40].

2.3.2. Neptune DB

Die *AWS Neptune DB* ist eine GDB, von der wie bei in Unterabschnitt 2.3.1 behandelten EC2-Containern, Instanzen gehostet werden können, die stündlich bezahlt werden und komplett in das AWS-Ökosystem eingebunden sind [42].

Neptune kann mit SPARQL oder Gremlin abgefragt werden [28] und das wiederum entweder über verschiedene Datenbankadapter für verschiedene Programmiersprachen, wie Python, Node.js oder Java, oder direkt per HTTPS mit SPARQL-, bzw. Gremlin-Abfragen [43].
Weiter ist völlige Verschlüsselung der Daten im gespeicherten Zustand und während dem Senden gewährleistet [44, 45]. Hierbei ist erstere durch ein Verschlüsseln der Daten auf der Festplatte sichergestellt [44] und letztere durch das Verschlüsseln mit HTTPS [45].
Außerdem bietet Neptune das automatische Erstellen von Datenbank-Backups an, um so die Datensicherheit vollständig zu gewährleisten [46].

3. Planungsphase

Im Folgenden werden notwendige Planungsschritte durchgeführt, um die Umsetzung dieses Projekts zu gewährleisten. Intern wurde diese Integration und dieses Projekt *Proteus* (Mond vom Planeten Neptun [47]; Gott der griechischen Mythologie [48]) getauft und soll von nun an diesen Namen tragen. Zunächst müssen in diesem Kapitel der Konzeption die Anforderungen klar definiert werden, auf deren Basis sich dann ein Entwurf für die Architektur des Projektes - sowohl nach außen hin zur Digital Heroes Platform (DHP), als auch im inneren von *Proteus* selbst - erstellen lässt und eine Technologieauswahl vorgenommen werden kann. Weiter müssen aus der Architektur hervorgehende Bedingungen und Einschränkungen definiert werden, unter denen *Proteus* laufen kann und muss.

3.1. Anforderungen

Die folgenden funktionalen und nicht-funktionalen Anforderungen werden an dieses Projekt gestellt, um eine Integration gewährleisten zu können.

1. Die Digital Heroes Platform (DHP) muss folgende *Create, Read, Update, Delete (CRUD)*-Operationen ausführen können:
    - Create
        - Neue *Heroes* mit den relevanten Eigenschaften (Specifics).
        - Neue *Missionen* mit den relevanten Eigenschaften (Sepcifics).
        - Neue *Ratings* von *Heroes* für die von ihnen beendeten *Missionen*.
    - Read
        - *Missions*-Vorschläge für die *Heroes*.
    - Update
        - Bestehende *Heroes* mit neuen Eigenschaften (Specifics).

  - Bestehende *Missionen* mit neuen Eigenschaften (Specifics).
  - Bestehende *Ratings* mit neuen Rating-Werten.
- Delete
  - Bestehende *Heroes* und deren Eigenschaften (Specifics).
  - Bestehende *Missionen* und deren Eigenschaften (Specifics).
  - Bestehende *Ratings*.

2. Danos muss in der Lage sein, über eine Go-Bibliothek folgende CRUD-Operationen auszuführen:
   - Read
     - *Heroes* und deren Eigenschaften.
     - *Missionen* und deren Eigenschaften.
     - *Rankings* von *Missionen*.
   - Create & Update & Delete:
     - *Freundschaften* zwischen *Heroes* und *Missionen*.
3. Die Bibliothek muss später eventuell noch weitere Operationen auf Grundlage der Daten der DHP für Danos anbieten.
4. Danos muss über Anfragen für *Missions*-Vorschläge mitsamt des angefragten *Hero* unterrichtet werden.
5. Danos muss unterrichtet werden, wenn eine der Bibliotheks-Funktionen/Methoden fehlschlägt.
6. Wird eine CRUD Operation der DHP nicht erfolgreich ausgeführt, muss die DHP über Fehlermeldungen darüber unterrichtet werden.
7. Die Daten von Danos und der DHP müssen persistent gehalten werden.
8. Die Daten der DHP dürfen auf keinen Fall für Dritte zugänglich sein und müssen DSGVO-konform verarbeitet und aufbewahrt werden.

9. Hohe modulare Erweiterbarkeit und Unterstützung für flexible Eigenschaften von *Missionen* und *Heroes*, da diese Eigenschaften noch nicht abschließend definiert sind.
10. Hohe Skalierbarkeit.
11. Die Danosintegration muss jeder Zeit, flexibel und von DHP-Admins abgeschaltet werden können.
12. Das Projekt muss auf AWS umgesetzt werden.

3.2. Architektur

In diesem Abschnitt wird die Architektur entworfen, die alle Anforderungen aus Abschnitt 3.1 erfüllt und bestmöglich in die bestehenden Systeme eingebunden werden kann. In diesem Abschnitt werden auch die Architektur- und Technologieentscheidungen im Bezug auf die Anforderungen erörtert.

3.2.1. Äußere Architektur (Blackbox)

Zuerst gilt es die äußere Architektur, beziehungsweise die Einbindung der Integration - *Proteus* - und der Danos-Engine in die bestehende *DHP* zu entwerfen.
Dabei ist eine der wichtigsten Voraussetzungen aus 3.1 die, dass *Proteus* auf der AWS Plattform laufen soll - dies war von den Danos-Entwicklern (Daniel Defiebre, Panagiotis Germanakos) so gewünscht worden und die SAP hat dieser Bitte stattgegeben. Mit diesem äußeren Rahmen stellt sich nun also die Frage, wie man die SAP Neo Cloud-Plattform auf der die DHP läuft (siehe Abschnitt 1.1) mit *Proteus* kommunizieren lässt. Hierfür ist zunächst noch wichtig, ob es sich um eine bidirektionale[1] oder unidirektionle[2] Kommunikation handelt. Aus den Anforderungen geht hervor, dass *Proteus* und Danos von der DHP nur Informationen benötigen, die die DHP selbst an *Proteus* übermittelt - *Proteus* oder Danos müssen also niemals die DHP anfragen, sofern diese das Versprechen (Promise) hält, *Proteus* immer über Änderungen der relevanten Daten zu unterrichten. Dieser Punkt

[1] Hier: Kommunikation die durch beide Kommunikationspartner *initiiert* werden kann.
[2] Hier: Kommunikation die nur durch **einen der beiden** Kommunikationspartner *initiiert* werden kann. Auf eine Anfrage des *Initiators* kann allerdings auch geantwortet werden.

wird also in die Dokumentation von *Proteus* und die Voraussetzungen aufgenommen. Mit dieser Einschränkung lässt sich allerdings ein unidirektionaler Informationsfluss etablieren, wodurch eine gegenseitige Abhängigkeit vermieden werden kann und *Proteus* keine Kenntnis über die DHP benötigt. Dies erleichtert auch eine lose Kopplung nach Abschnitt 3.1 Punkt 11, da somit nur auf der DHP eine Einstellung vorgenommen werden muss, um die Danos-Integration zu entfernen, und *Proteus* nicht ebenfalls über diese Entscheidung informiert werden muss. Außerdem muss sich bei einer unidirektionalen Kommunikation *Proteus* nicht auch noch bei der DHP authentifizieren, sondern nur die DHP bei *Proteus*, anstatt beide gegenseitig, was den Implementationsaufwand reduziert und wieder für die lose Kopplung spricht, da weniger Danos-spezifischer Code in der DHP eingebunden werden muss. Anforderungsgemäß werden bei Fehlern der Create, Read, Update, Delete (CRUD)-Operationen, diese an die DHP zurückgegeben.
Die nächste Anforderung die es zu betrachten gilt, ist die nach Abschnitt 3.1 Punkt 7 und Punkt 8 zur DSGVO-konformen, persistenten Datenhaltung. Um die Daten der DHP und die generierten Daten von Danos nicht nach Neustart, Stromausfall und wegen unzureichendem Arbeitsspeicher zu verlieren, muss eine nicht-volatile Speichermethode herangezogen werden. Dafür eignet sich eine Datenbank, beziehungsweise insbesondere eine Graph Database (GDB), da diese nach Unterabschnitt 2.2.1 technisch gesehen überlegen sind und da das Danos-Modell sowieso eine Graphstruktur aufweist (s. Abschnitt 1.2). Auch hier ist wieder eine unidirektionale Kommunikation zu verwenden - diesmal zwischen *Proteus* und der GDB - da die Datenbank keine Kommunikation initiieren können muss. Die DSGVO-Konformität lässt sich durch eine Verschlüsselung der Daten auf der GDB bewerkstelligen: Dies kann entweder durch die Datenbank selbst geschehen, oder durch eine Verschlüsselung mit *Proteus*. In jedem Fall macht es für die äußere Architektur keinen Unterschied.
Eine weitere Architekturentscheidung ergibt sich aus Abschnitt 3.1 Punkt 10, wonach *Proteus* in der Lage sein soll simultan auf vielen Computing-Instanzen zu laufen um tausende simultane Anfragen zu bearbeiten. *Proteus* wird so gebaut, dass dies möglich ist, ohne Dateninkonsistenzen zu haben, und eine einzige sogenannte *Source of Truth* - die GDB - zu haben. Um die vielen Anfragen optimal auf die einzelnen *Proteus*-Instanzen verteilen zu können, bedarf es einem Load-Balancer (Lastenverteiler), der den *Proteus*-Instanzen vorgeschaltet ist. Dieser Lastenverteiler könnte selbst implementiert werden, doch auch hier ist es (wie schon bei der Datenbank) einfacher, stabiler und sicherer, auf bereits bestehende Technologie zurückzugreifen. Dieser Lastenverteiler, der ebenfalls

die Definition eines *Reverse-Proxys* erfüllt, nimmt also - wie der Name schon sagt - als umgekehrter *Stellvertreter* die gesamten DHP-Anfragen entgegen und verteilt sie auf die *Proteus*-Instanzen. Die *Proteus*-Instanz überprüft dann, ob die Anfrage authentifiziert und autorisiert ist und verarbeitet dann gegebenenfalls die Anfrage weiter und schickt die Antwort wieder über den Reverse-Proxy zurück zur DHP. Somit macht es auch nur Sinn, dass dieser Stellvertreter die Verschlüsselung im Transit von den DHP-Daten, über das Internet, zu dem AWS Reverse-Proxy übernimmt. Natürlich müssen die Daten auch im Transit vom Reverse-Proxy zu den einzelnen *Proteus*-Instanzen verschlüsselt werden, was auch der Fall sein wird. Dies wird durch die Auswahl an Technologien sichergestellt. Abbildung 3.1 (Unterabschnitt 3.2.2) zeigt eine schematische Darstellung der äußeren Architektur, wozu allerdings gesagt sei, dass hier bereits die verwendeten Technologien dargestellt sind, die im Folgenden erörtert werden:

3.2.2. Technologieauswahl

In Unterabschnitt 3.2.1 wurde bereits die technologie-unabhängige[3] äußere Architektur festgelegt, die jetzt mit Leben gefüllt wird:
Hierfür wird zuerst die Kommunikationstechnologie, beziehungsweise das -protokoll festgelegt. Zur Kommunikation zwischen der DHP und AWS gibt es mehrere mögliche Technologien wie zum Beispiel *gRPC*, was ein open-source *Remote Procedure Call (RPC)*-System darstellt, was über ein Netzwerk Methoden und Funktionen in anderen Programmen aufrufen kann, und deren Ergebnisse zurückliefert [49]. Eine weitere Technologie wäre das *Hypertext Transfer Protocol (HTTP)*, beziehungsweise die verschlüsselte Variante *HTTP Secure (HTTPS)*. Das erstere benötigt allerdings eine Installation und Setup auf beiden Seiten - sowohl auf der der DHP, sowie auf der von AWS. Da allerdings nur recht schwer und über viele Umwege auf des Environment von der Neo-Cloud Einfluss genommen werden kann, ist die Möglichkeit gRPC zu verwenden sehr aufwendig und eher ungünstig. Dagegen wird *HTTPS* bereits in der DHP verwendet, ist simpel und hat native Unterstützung von Java selbst [50]. Damit ist auch klar, dass die *Representational State Transfer (REST)*[4] *API*, die die DHP bedienen wird, eine HTTP REST

[3] Natürlich bis auf die Annahme, dass das Umfeld *AWS* angenommen wurde, was ebenfalls eine Anforderung ist.

[4] REST weil die API so aufgebaut wird, dass sie alle CRUD-Operationen standardkonform implementieren wird.

API sein wird. Während die Grundversion HTTP sehr simpel wäre, unterstützt sie keine Verschlüsselung, die allerdings unbedingt benötigt wird. Dies wird es auch nötig machen auf *Proteus* Seite *Secure Sockets Layer (SSL)*-Zertifikate anzubieten, da diese ein Teil des Verschlüsselungsprozesses sind [51]. Diese Zertifikate können zwar selbst erstellt werden, dann tragen sie aber nicht das Siegel einer *Certificate Authority (CA)* und werden von Webbrowsern (und auch der *Java Virtual Machine* [52]) als unsicher eingestuft [51]. Im Falle der Java-Bibliothek wird dann auch gar keine Verbindung aufgebaut [52]. Somit werden vertrauenswürdige SSL-Zertifikate notwendig, die dem Reverse-Proxy zur Verfügung stehen. Die CA *Let's Encrypt* bietet kostenlose, kommerziell-nutzbare Zertifikate an [53]. Für dieses Projekt werden deswegen diese verwendet. Mit der Software *Certbot* ist es außerdem möglich die benötigten Zertifikate einfach und nach Ablauf der Zertifikate vollautomatisch zu installieren, sodass sie für den Reverse-Proxy nutzbar sind [54].
Die Kommunikation von Reverse-Proxy und *Proteus* muss dann ebenfalls über HTTP stattfinden, sodass der Reverse-Proxy keine teure Protokoll-Konvertierung vornehmen muss. Hier reicht allerdings die unverschlüsselte Version genau dann, wenn die Kommunikation trotzdem verschlüsselt ist. Dies kann verwirklicht werden, indem man die kostenfreie *Virtual Private Cloud (VPC)*-Technologie von AWS nutzt. Hiermit werden alle AWS-Komponenten in ein verschlüsseltes, virtuelles, privates Netzwerk eingegliedert [55, 56]. Somit kann der HTTPS-Verschlüsselungs-Overhead gespart werden und die Daten sind trotzdem im Transit verschlüsselt [55].
Als Format der Datenübertragung per HTTP(S) wird der de facto Industriestandard *JavaScript Object Notation (JSON)* gewählt.
Zur Auswahl der Graph Database (GDB): Es liegt natürlich Nahe die AWS-hauseigene GDB Neptune zu verwenden und das ist auch das sinnigste. Neptune wird gehostet, liegt im AWS-Netzwerk, was super schnelle Kommunikation ermöglicht, und unterstützt Verschlüsselung der Daten sowohl im Transit, als auch abgespeichert [44, 45]. Neptune benutzt zur Kommunikation HTTPS[5], was nicht ganz optimal ist, da es in der eigenen VPC unnötigen Verschlüsselungs-Overhead - die Daten sind nun doppelt verschlüsselt - hinzufügt. Dafür ist die Authentifizierung in der eigenen VPC sehr komfortabel, da Anfragen an Neptune aus dem selben VPC ohne weitere Authentifizierung bearbeitet werden. Das ist auch der hauptsächliche Grund, eine VPC einzurichten. Somit ist der Neptune-Service auch gar nicht von außen ansprechbar, was wiederum die Sicherheit der

[5]Neptune kann auch von außerhalb des verschlüsselten VPC-Umfelds erreichbar sein, soll es aber in diesem Projekt nicht.

Daten erhöht. Im Bezug auf Neptune wird Gremlin als Query-Sprache gewählt.
Weiter drängt sich die kostenlos kommerziell-nutzbare Open-Source-Software Nginx[6] als Reverse-Proxy und Load-Balancer geradezu auf [57]. Nginx wird von Certbot unterstützt, was das SSL-Zertifikats-Management nochmals erleichtert [54], und Nginx ist eine zukunftssichere Technologie als der populärste Webserver [58]. Wegen der guten Zukunftsaussichten, der großen Userbase und der guten Performanz wird Nginx der Reverse-Proxy/Load- Balancer der Wahl für dieses Projekt.
Als Server bzw. Computing-Instanz für die *Proteus*-Instanzen, sowie für den Nginx Reverse-Proxy werden jeweils EC2-Instanzen verwendet, aufgrund er in Unterabschnitt 2.3.1 beschriebenen Eigenschaften, wie z. B. extrem einfache Skalierbarkeit.
Als Programmiersprache für *Proteus* selbst wird Go verwendet, da Danos in Go geschrieben ist (s. Abschnitt 1.2) und *Proteus* eine Software-Bibliothek für Danos bieten muss. Es wäre möglich einzelne Komponenten in anderen Programmiersprachen zu verfassen, aber da es in der Abteilung sowieso Go-Entwickler gibt und derjenige Kollege, der nach meinem Abschied *Proteus* warten wird, Go-Entwickler ist, macht das keinen Sinn.

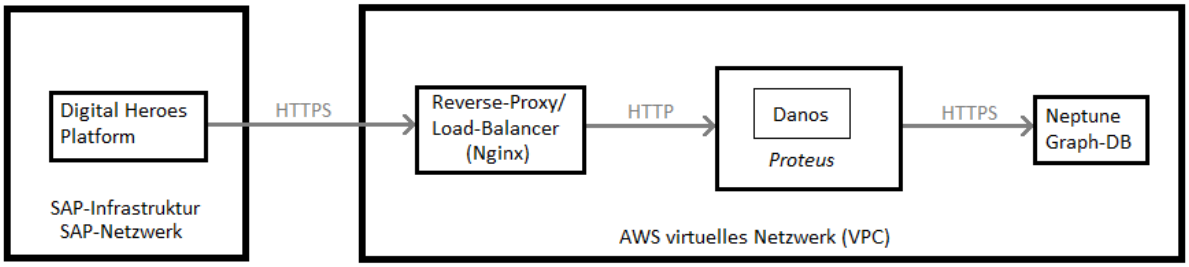

Abbildung 3.1.: Äußere Architektur (Blackbox) von *Proteus*. Hier können auch noch weitere parallele *Proteus*-Instanzen eingefügt werden.
(Abb. entnommen aus eigener Arbeit)

3.2.3. Innere Architektur (Whitebox)

In diesem Abschnitt wird nun die innere Architektur von *Proteus* selbst betrachtet. Beim Entwurf muss unter Anderem darauf geachtet werden keine zyklischen Abhängigkeiten der einzelnen Softwarekomponenten zu erzeugen.
Außerdem muss beachtet werden, dass die selben Daten in vielen unterschiedlichen Formaten vorliegen müssen. So benötigt HTTP das JSON-Format, während Danos

[6] Gesprochen: *Engine-X* [57].

verschiedene Go-Strukturen verwendet, und Gremlin für Neptune wieder besonders serialisiert werden muss, um die Daten ein- und auszulesen. Während das den Einsatz von verschiedenen Parsern erforderlich macht, benötigen einige Teile der Software bestimmte Graphstrukturen (Knoten und Kanten), die in dem Go-Package[7] *Graph* zusammengefasst sind.
Weiter benötigt *Proteus* einen Web-Server für die HTTPS-Requests von der DHP. Diese werden im Package *Gatekeeper* entgegengenommen und zum Package *Cache* weitergereicht, welches den - nun ja - Cache der Daten aus Neptune hält und aktualisiert; sowie ebenfalls durch das Package *Gremlin*, welches den Datenbankadapter für Neptune darstellt, eine Aktualisierung in Neptune selbst anstößt. Wird eine Aktualisierung der Daten durch Danos vorgenommen, so kommt dies auch erst zum Package *Cache*, um zu überprüfen, ob eine Aktualisierung des Caches notwendig wird und wird dann über *Gremlin* wieder an Neptune weitergeleitet.
Gatekeeper unterrichtet Danos über Anfragen zum Missionsvorschlag über das *Befehl*-Design-Pattern, was es ermöglicht, dass *Gatekeeper* keine Kenntnis von Danos benötigt. So wird eine Go-Funktion an Gatekeeper übergeben, die dann bei Zugriff auf den REST-API-Endpoint aufgerufen werden kann, ohne dass *Gatekeeper* eine Abhängigkeit auf Danos hat.
Letztlich gibt es noch das *Proteus*-Package, welches lediglich per *Proxy*-Design-Pattern als Interface für Danos fungiert, zur einfachen Initialisierung genutzt wird, und die anderen APIs der vorgenannten Packages vor Danos versteckt. Von *Proteus* erhält Danos ein Go-Interface, um mit der Datenhaltung (Cache, DB) zu interagieren. In Abbildung 3.2 sind schematisch die Abhängigkeiten der einzelnen Packages voneinander durch schwarze Pfeile dargestellt. Graue Pfeile stellen eine Kommunikation mit anderen Micro-Services dar.

Ein weiterer wichtiger Aspekt ist die Autorisierung und Authentifizierung für den Web-Server *Gatekeeper*. Hierbei wird mit API-Keys und einem Token-System gearbeitet. Da es nur wenige, vordefinierte Nutzer des Web-Servers mit bestimmten Berechtigungen gibt, wird hierfür keine separate Datenbank nötig, sondern die API- Keys sind in einer *Permissions*-Datei[8] anzulegen und für die *Proteus*-Instanzen zugänglich zu machen (z.B. über *Network Attached Storage (NAS)*). Der Inhalt dieser Datei wird dann beim Starten des Web-Servers einmalig in den Arbeitsspeicher geladen. Die API-Keys werden hier

[7] Vgl. Java-Packages
[8] Definition der Semantik und der Syntax befinden sich in der Dokumentation von *Proteus*

definiert als eine Kombination aus ID (vgl. Username) und Secret (vgl. Passwort) und werden per *Basic Auth* mit HTTPS an den Server übertragen [59]. Das Secret wird gehashed und mit dem geladenen Inhalt der Permissions-Datei abgeglichen. Außerdem wird hier noch die Rolle überprüft, um zu schauen, ob der übermittelte Key Erlaubnis hat die Ressource aufzurufen. Bisher gibt es keine Ressourcen, die nur für bestimmte Keys zugelassen sind, aber das könnte sich in der Zukunft ändern und wird deswegen bereits berücksichtigt.

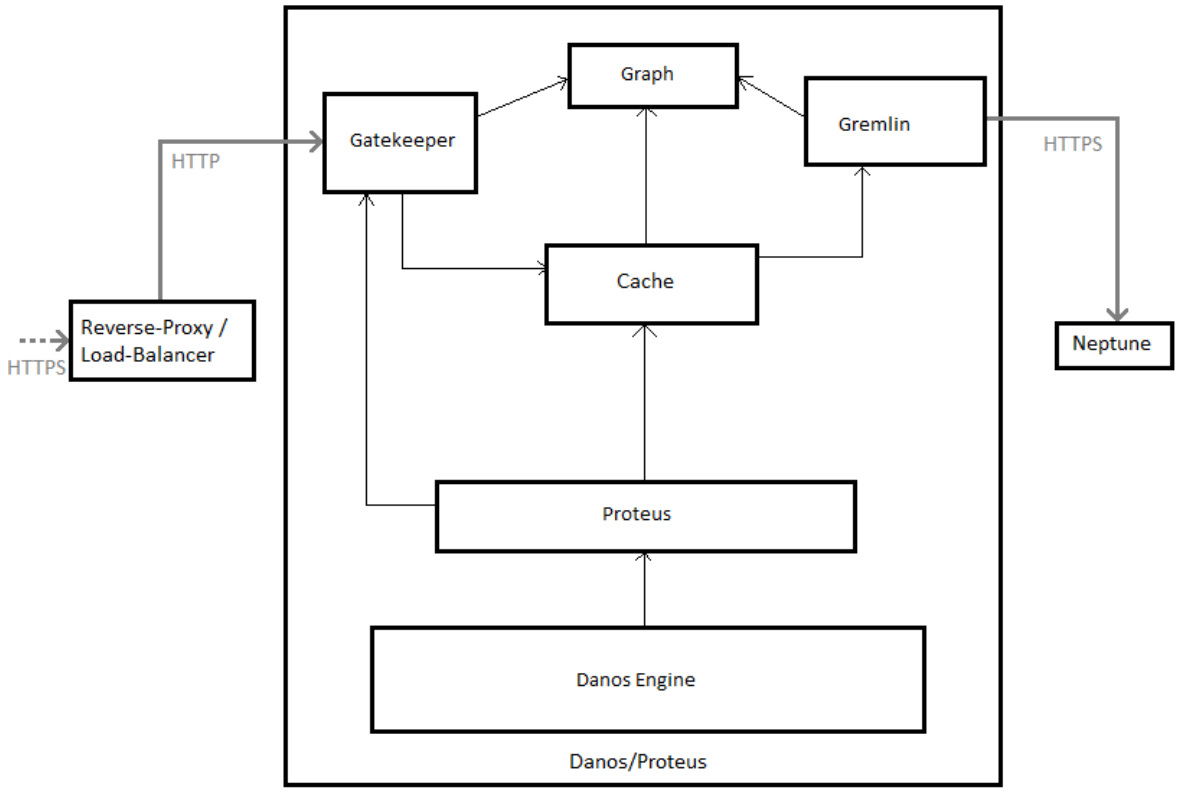

Abbildung 3.2.: Innere Architektur (Whitebox) von *Proteus*.
(Abb. entnommen aus eigener Arbeit)

3.3. Voraussetzungen

Nach Unterabschnitt 3.2.1 und 3.2.2 müssen bestimmte Umgebungsvoraussetzungen gegeben sein, damit dieses Projekt problemlos funktioniert. Die vorgeschlagenen Tech-

nologien für den Reverse-Proxy/Load-Balancer und die SSL-Zertifikate/CA aus Unterabschnitt 3.2.2 müssen nicht unbedingt eingehalten werden. Die Neptune DB, sowie die Protokolle hingegen schon. Einige der folgenden Voraussetzungen könnten theoretisch von diesem Projekt selbst umgesetzt werden, sind aber aufgrund des unverhältnismäßigen zusätzlichen Aufwandes sowie der bessern Microservice-Architektur in externe Services oder Umgebungskonfigurationen ausgelagert worden:

1. Es muss eine *AWS Virtual Private Cloud (VPC)* - ein verschlüsseltes virtuelles Netzwerk [55, 56] - auf AWS geben, in dem enthalten sind:
   - Eine EC2-Instanz mit diesem Projekt und der implementierenden Danos-Engine
   - Eine AWS Neptune DB Instanz
   - Eine EC2 Instanz mit einem Reverse-Proxy/Load-Balancer
2. Die Neptune DB Instanz muss Zugriffe aus dem eigenen virtuellen Netzwerk (VPC) zulassen, oder Zugriffe durch den EC2-Container mit der Danos-Engine und diesem Projekt selbst, da es keine weitere Authentifizierung und Autorisierung von diesem Projekt für die Neptune-Instanz geben wird, außer über einen Zugriff aus dem eigenen virtuellen Netzwerk (VPC).
3. Es muss ein Reverse-Proxy/Load-Balancer innerhalb des virtuellen Netzwerkes (VPC) existieren, der den HTTPS- verschlüsselten Traffic der DHP entgegennimmt und dann per HTTP zum EC2-Container mit der Danos-Engine und diesem Projekt weiterleitet. Dies ist völlig sicher, da der Traffic im virtuellen Netzwerk (VPC) verschlüsselt ist [55] (vgl. Virtual Private Network (VPN)).
4. Es muss eine Domain geben, die auf den EC2-Container mit dem Reverse-Proxy/Load-Balancer verweist.
5. Der Reverse-Proxy/Load-Balancer muss aktuelle und von einer offiziellen CA gezeichnete SSL-Zertifikate verwenden.
6. Es muss eine *Permissions*-Datei in dem virtuellen Netzwerk (VPC) liegen, die gemäß Spezifikation zur Benutzerauthentifikation und -autorisierung genutzt werden kann.
7. Derjenige, der die Danos-Engine mithilfe diesen Projektes implementiert, muss über den Pfad zur *Permissions*-Datei, die Neptune-Endpoint-Adresse, und den

Weiterleitungsport des Reverse-Proxys/Load-Balancers informiert werden. Diese Informationen können beispielsweise als Umgebungsvariablen definiert werden.

Anmerkung: Der Reverse-Proxy/Load-Balancer und die Danos-Engine können auch auf dem selben EC2-Container liegen.

Letztlich gilt es noch hinzuzufügen, dass die DHP in der Pflicht steht *Proteus* zu updaten solange es integriert sein soll, sodass die Architektur aus Unterabschnitt 3.2.1 funktionieren kann.

4. Implementierung

Im Folgenden werden die wichtigsten Elemente der Implementierung genauer erläutert und vorgezeigt. Hierzu gehört zunächst die *Proteus*-API und der Cache, der gleichzeitig die Danos-API, sowie die Pipeline von der REST API zur Datenbank darstellt. Dann wird ein genauerer Blick auf die REST API zusammen mit der Authentifizierung/Autorisierung geworfen, und schließlich noch der Datenbankadapter mit Gremlin genauer betrachtet.

4.1. Exponierte API und Cache

In diesem Abschnitt wird die exponierte API *Proteus*[1] und der Cache, durch den alle Daten fließen, behandelt.

4.1.1. Cache

Da *Gatekeeper* und Danos unterschiedliche Daten manipulieren können sollen (vgl. Abschnitt 3.1), macht es nur Sinn den Cache in Form zweier Interfaces zu exponieren, wovon *Gatekeeper* und Danos jeweils eines bekommen. Abbildung 4.1 zeigt ein Klassendiagramm mit dieser gekürzten Hierarchie der Implementierung. Die implementierende Struktur *cache*[2] ist privat in dem gleichnamigen Package und hält die gesamten im Cache befindlichen Daten als Attribute zusammen mit einer *gremlin.Query*-Struktur, die als Datenbankadapter dient und einem *sync.Mutex*, welcher genutzt wird, um verschiedene parallele Ausführungsfäden - Goroutines - zu synchronisieren.
Das *Api*-Interface stellt die Methoden für Danos bereit, um verschiedene Query-Operationen durchzuführen. Einer dieser Methoden ist *GetFriendshipCount*. Sie nimmt die ID eines Elements als Zeichenkette und gibt die Anzahl der Danos-*Freundschaften* für das per ID bestimmte Element als Ganzzahl zurück.
Das *Pipeline*-Interface wird von der REST API verwendet und stellt Methoden bereit, die

[1] Hier ist damit das Go-Package gemeint und nicht das ganze Projekt. Siehe Abbildung 3.2.
[2] Private Strukturen in Go werden klein geschrieben

es erlauben die Datenbank und den Cache zu manipulieren. So sind einige dieser Methoden direkte Weiterleitungen der CRUD-Operationen aus der REST API zusammen mit einigen zusätzlichen Getter-Methoden, wie *VertexExists*, um ein besseres Error-Reporting auf Seiten der REST API zu ermöglichen und aussagekräftige Fehlermeldungen an den User (DHP) übermitteln zu können.
Die cache-Struktur hat natürlich alle Methoden der Interfaces und implementiert diese, um die Interfaces zu implementieren. Eine solche beispielhafte Implementierung lässt sich in Code-Ausschnitt 4.1 sehen.

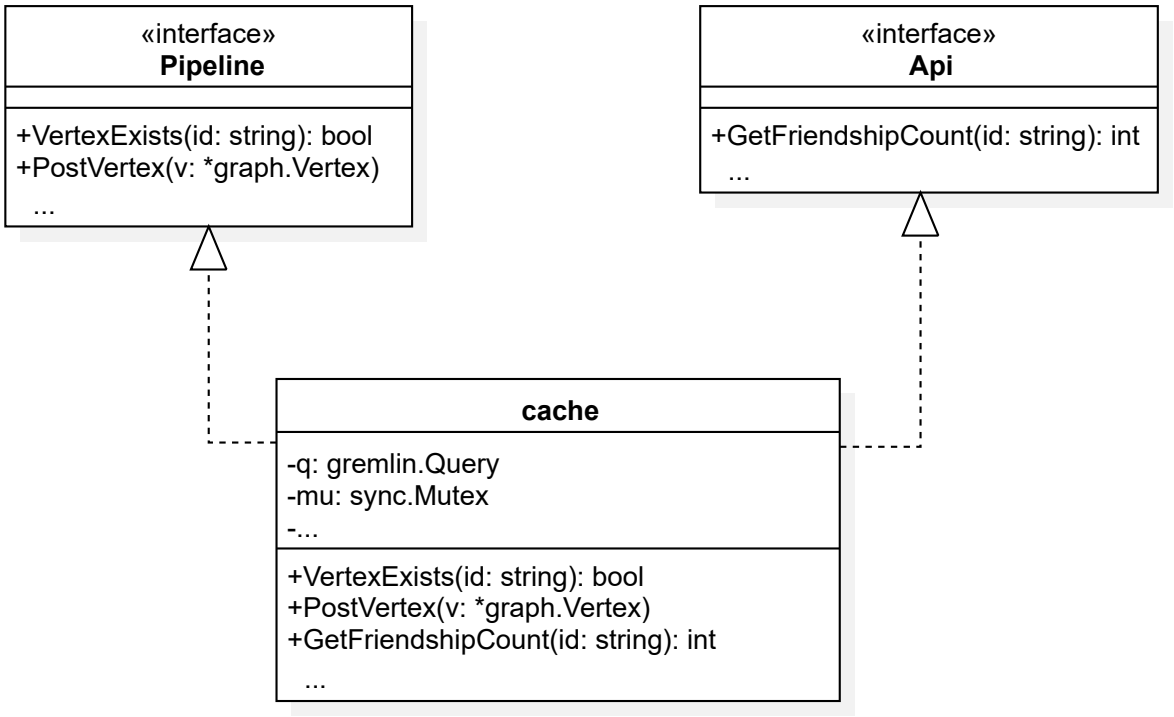

Abbildung 4.1.: Implementierungshierarchie von *Cache* und dessen *Interfaces* (gekürzt). (Abb. entnommen aus eigener Arbeit)

In Code-Ausschnitt 4.1 implementiert die *cache*-Struktur per Receiver-Methode die Interfacemethode *VertexExists*. Dieser Ausschnitt ist optimal, da einige grundsätzliche Funktionsweisen erkennbar sind. Die Methode soll erkennen, ob ein Knoten jeglicher Art (egal ob Hero oder Mission) mit der gegebenen ID bereits existiert. Dies ist wichtig, da die IDs von Heroes und Missionen eindeutig sein sollen. Zur Übersichtlichkeit und aufgrund unfertiger Implementierung des Überprüfens des Caches, aufgrund von fehlender Definition von einigen Knoten seitens des Danos-Teams, wird die Überprüfung des Caches

hier ausgelassen. Die Methode beginnt und muss aufgrund von Multithreading (um Race Conditions zu vermeiden) den Mutex schließen und am Ende der Methode wieder öffnen (`defer` führt eine Methode immer am Ende der aufrufenden Methode auf, egal durch welchen Pfad). Solange der Mutex geschlossen ist, müssen alle anderen Methoden, die versuchen den Mutex zu schließen, warten, bis er wieder geöffnet wurde. Das heißt, in den meisten *cache*-Methoden wird der Mutex geschlossen, um so die Zugriffe auf den Cache zu serialisieren.
Als Beispiel: Angenommen die *cache*-Zugriffe wären nicht serialisiert und es kämen zwei kurz-aufeinander-folgende Anfragen zur REST API, wobei die erste eine den Knoten löscht und die zweite versucht einen Knoten mit dieser (theoretisch) neu-freigewordenen ID per *POST* zu erstellen: Zuerst kommt die *DELETE*-Anfrage an, wird aber noch nicht ganz abgeschlossen, bis der Scheduler entscheidet zur POST-Anfrage zu wechseln. Hier wird überprüft ob der Knoten bereits existiert, was er noch tut, dann wird die POST-Methode unterbrochen und DELETE darf fortfahren und der Knoten wird gelöscht. DELETE liefert an die DHP eine erfolgreiche Löschung zurück, POST liefert allerdings zurück, dass die ID bereits existiert. Der Nutzer wird also nicht wissen, welche der beiden Methoden fehlgeschlagen ist, da die eine sagt, dass die ID gelöscht wurde, aber die andere sagt, dass die ID noch existiert. Wird jetzt allerdings der Mutex geschlossen kommt die DELETE-Anfrage zuerst dran (kommt zuerst an) und schließt die Barriere. Jetzt kommt die POST-Anfrage dran, kann allerdings nicht die Existenz des Knotens abfragen, weil der Mutex geschlossen ist. Die DELETE-Methode ist wieder am Zug, löscht den Knoten, öffnet den Mutex. Jetzt darf die POST-Methode wieder, schließt den Mutex für sich, erstellt den Vektor und alles funktioniert wie erwartet trotz Parallelität.
Die VertexExists-Methode steuert also den Mutex an, überprüft dann den Cache auf die Existenz des Knoten. Wenn dort nichts gefunden wurde, überprüft es die Datenbank. Dies geschieht mit der Methode *GetVertex* von der Query q des caches c.

```
1 func (c *cache) VertexExists(id string) (bool, error) {
2     defer c.mu.Unlock()
3     c.mu.Lock()
4
5     // Caching goes here
6
7     _, err := c.q.GetVertex(id)
8     if err == gremlin.ErrNotFound {
```

```
 9         return false, nil
10     } else if err != nil {
11         return false, err
12     }
13     return true, nil
14 }
```

Code-Ausschnitt 4.1: Existenzüberprüfung eines bestimmten Knoten nach ID.

4.1.2. Exponierte Proteus-API

Der für Danos exponierte Teil der API hat nur eine kleine Zahl an Methoden, die da wären:

- `Initialize(...)` - Initialisiert das Programm mit einigen der nach Abschnitt 3.3 definierten Umgebungsvariablen und der in Unterabschnitt 3.2.3 genannten Callback-Funktion, die nach dem Befehl-Design-Pattern arbeitet.
- `StartRestApi(addr: string)` - Startet den REST API Service, der auf den angegebenen Port hört, in einer neuen Goroutine.
- `GetCache()` - gibt den Cache-Singleton als *Api*-Interface zurück, welchen Danos dann benutzen kann um weiter mit *Proteus* zu interagieren.

In Code-Ausschnitt 4.2 wird der Pfad zur Permission-Datei, sowie der Neptune- Endpoint übergeben. Auch wird hier der *Callback* übergeben, mit dem Danos über Anfragen zur Ermittlung von Missionsvorschlägen benachrichtigt wird. Die Funktion muss eine ID für einen Hero entgegennehmen und kann dann eine Liste von Zeichenketten, die die vorgeschlagenen Missions-IDs nach Relevanz sortiert modellieren sollen, und einen möglichen Fehler zurückgeben.
Die Methode *Initialize* initialisiert daraufhin den *cache*, welcher einige kurze Basistests zur Datenbankverbindung durchführt und einen Fehler zurückgibt, wenn damit etwas nicht stimmt. Danach wird der Web-Server initialisiert und verhält sich analog zur Initialisierung des *cache*s, wenn die Daten zur Authentifizierung und Autorisierung, wie in Unterabschnitt 3.2.3 beschrieben, nicht geladen werden können. Hier werden unter anderem auch der *cache*- Singleton als *Pipeline*, und der *Callback* an den Web-Server übergeben. Letztlich wird eine globale Variable gesetzt, die vermerkt, dass die

Initialisierung erfolgreich stattgefunden hat und die anderen Funkionen nun verwendet werden können.

```
1  func Initialize(permissionFilePath string, neptuneEndpoint ↙
       ↳ string, callback func(id string) ([]string, error)) {
2      if err := cache.Initialize(neptuneEndpoint); err != nil {
3          log.Fatal("fatal error: db connection is not properly ↙
               ↳ working with endpoint", neptuneEndpoint,
4              "error message:", err)
5      }
6      if err := gatekeeper.Initialize(cache.GetCache(), ↙
           ↳ permissionFilePath, callback); err != nil {
7          log.Fatal("necessary files for authentication could not ↙
               ↳ be loaded correctly:", err)
8      }
9
10     inited = true
11 }
```

Code-Ausschnitt 4.2: Initialisierung des *Proteus*-Services.

4.2. REST API

In diesem Kapitel ist vor allem die Authentifizierung und Autorisierung von großem belangen und macht ein gutes Stück des Codes und der Herausforderung der Integration aus. Daneben wird natürlich auf die Request-Handler-Funktionen selbst ein Blick geworfen.

4.2.1. Authentifizierung und Autorisierung

Zur Überprüfung der Identität wird jede Request durch eine Middleware geführt, die diese Überprüfung vornimmt, bevor dann die eigentlichen Requests ausgeführt werden. Aber vorher ist das Format der in Unterabschnitt 3.2.3 und Abschnitt 3.3 genannten *Permissions*-Datei zu definieren.
In Code-Ausschnitt 4.3 ist die auch in der Dokumentation enthaltene Beispiel-Permissions-Datei zu sehen, nach der diese Dateien aufgebaut werden müssen. Die JSON-Datei ist

folgendermaßen aufgebaut: Die einzelnen API-Keys sind in einem Array angeordnet, wobei jeder Key wiederum aus einer ID, einem Secret, und einem Array von Berechtigungen besteht. Die ID ist beliebig wählbar. Das Secret muss in die Datei als der hexadezimal-kodierte SHA1-Hash des eigentlichen Secrets eingetragen werden. Das Secret sollte zufällig generiert werden, Sonderzeichen, Groß- und Kleinbuchstaben, und Zahlen enthalten, und 30-50 Zeichen lang sein. Ein Beispiel wäre das generierte Secret `xZFj&#3BtR9Gj^ddbjCsuKDCin$Q3` dessen hexadezimal- kodierter SHA1-Hash `71e5fa01f3e2951147859b0408385b6a8cdd7400` in die Datei eingetragen werden müsste. Es wurde die kryptographische Hash-Funktion SHA1 gewählt, weil sie deutlich schneller ist als eine sicherere Alternative wie Bcrypt (durch Benchmarks getestet). Hier stellen wir Schnelligkeit über Sicherheit, weil die Keys allgemein auf einem sehr sicheren System liegen und weil die Secrets durch kompetente Systemadministratoren nach den genannten Anforderungen generiert werden, was einen Brute-Force Angriff auf die Keys sehr schwierig machen sollte, was zwar begünstigt wird durch die schnelle Hash-Zeit, aber wegen der sicheren Secrets, kein Bedenken sein sollte. Außerdem sind zu viele konsekutive Zugriffe auf das System mit einer IP oder auch einem Botnet durch AWS abgesichert [60]. Zuletzt gibt es noch ein Array für die Berechtigung für User mit diesem API-Key. Ein leeres Array hat keine Berechtigungen und kann sich also nur authentifizieren, nicht aber autorisieren. Die leere Zeichenkette impliziert die Basis-Berechtigungen. Alle weiteren Einträge geben Zugriff auf zusätzliche Endpoints oder Services. Bisher gibt es nur Endpoints die mit Basis-Berechtigungen erreichbar sind.
In Abbildung 4.2 ist vereinfacht der Ablauf der Authentifizierung und Autorisierung dargestellt. Dazu sei gesagt, dass der *authenticator* passende Meldungen erzeugt und den Request vorzeitig mit dieser Meldung an die DHP zurück gibt, wenn der gegebene API-Key nicht formal korrekt ist (unmöglich zu parsen), oder wenn der Key keinem der geladenen Keys gleicht, oder wenn der Key gefunden wurde, dieser aber nicht die nötigen Berechtigungen hat. Die API-Keys werden per HTTPS Basic Auth übermittelt, weshalb die Transmission sämtlicher Anfragen (auch der ersten) über das Internet streng verschlüsselt sein müssen [59].

```
{
  "keys": [
    {
      "id": "userid",
      "secret": "SOME_SHA1-HASHED_HEX-ENCODED_SECRET",
      "permission": ["", "extended_permission_1"]
    },
    {
      "id": "useridbasic",
      "secret": "SOME_OTHER_SHA1-HASHED_HEX-ENCODED_SECRET",
      "permission": [""]
    }
  ]
}
```

Code-Ausschnitt 4.3: *Proteus* *Permissions*-Datei-Template mit API-Keys und Berechtigungen.

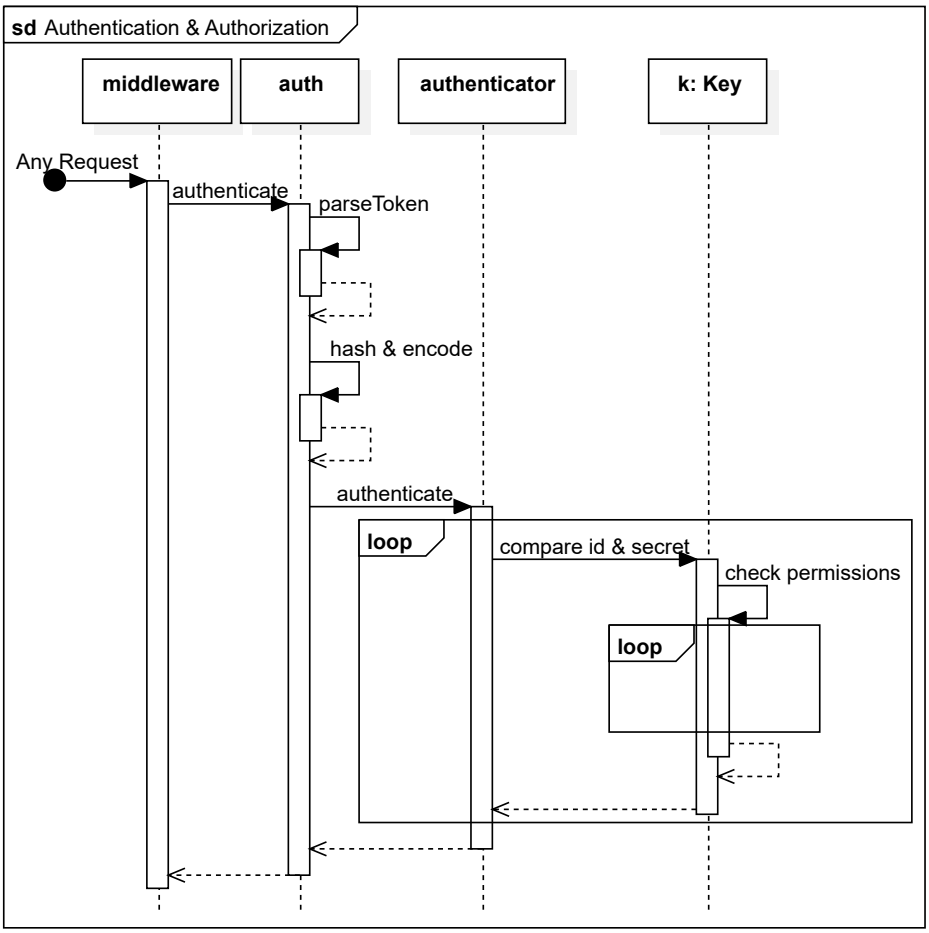

Abbildung 4.2.: Sequenzdiagramm zum Ablauf eine Authentifizierung und Autorisierung (vereinfacht).
(Abb. entnommen aus eigener Arbeit.)

4.2.2. Request-Handler

Im Folgenden werden die Request-Handler implementiert und da sich die Handler alle sehr ähneln, reicht ein Beispiel um die grobe Vorgehensweise für alle anderen Handler aufzuzeigen.
Die Funktion die in Code-Ausschnitt 4.4 dargestellt ist wird direkt von dem POST-Hero-Handler und POST-Mission-Handler gerufen, mit dem Unterschied, dass als Label *Hero* beziehungsweise *Mission* übergeben wird.
Die Funktion versucht zuerst aus dem Request-Body, der sich im JSON-Format befindet

(sichergestellt durch Middleware), den Knoten für die Graphdatenbank zu parsen. Die Definition für die Struktur und das Format der JSON sind in der Dokumentation dokumentiert, besteht aber aus zwei Feldern, der ID als Zeichenkette und den Eigenschaften (Properties) als weitere Struktur mit beliebig-betitelten weiteren Feldern. Daraufhin wird das Label für die Vertexstruktur gesetzt, was effektiv den Typ bestimmt. Dann sollte bei einer POST-Request überprüft werden, ob die ID bereits existiert. Sollte das der Fall sein, so sollte POST nicht die bisherige Ressource mit dieser ID überschreiben, sondern einen Fehler zurückgeben, was auch geschieht. Zuletzt wird der neue Knoten über die *Pipeline* zum Cache geschickt, der die Struktur dann in die Datenbank einpflegt. Geht hier nichts schief, so wird der HTTP-Statuscode `201 Created` geschrieben, was dem Nutzer suggeriert, dass die Operation erfolgreich durchgeführt wurde.

```
1  func postVertex(w http.ResponseWriter, r *http.Request, label ↙
       ↳ string) {
2      v, err := parseVertex(r.Body, true)
3      if err != nil {
4          http.Error(w, err.(*httpError).msg, err.(*httpError).code)
5          return
6      }
7
8      v.Label = label
9
10     exists, err := pipe.VertexExists(v.Id)
11     if err != nil {
12         log.Println(err)
13         http.Error(w, "", http.StatusInternalServerError)
14         return
```

Code-Ausschnitt 4.4: Funktion die bei POST von Heroes und Missionen aufgerufen wird. Label bestimmt den Typ.

4.3. Datenbankadapter

Für den Datenbankadapter ist wichtig, dass er die Zugriffe auf die Datenbank erleichtern soll. Außerdem soll hier die ganze datenbankspezifische Logik liegen. Dazu gehören neben

den erforderlichen Parsern zwischen Datenbankkonstrukten und Go-Konstrukten auch die in Unterabschnitt 2.2.3 untersuchten Gremlin-Queries. Hierzu ist die interne Darstellung eines Knoten interessant. Dieser enthält folgende Felder:

```
type Vertex struct {
    Label    string
    Id       string
    Properties map[string]string
}
```

Code-Ausschnitt 4.5: Interne Darstellung eines Knoten des Packages *Graph*.

4.3.1. Parser

Beginnend mit den Parsern muss gesagt werden, dass Neptune auf die HTTP-Gremlin-Queries im JSON-Format antwortet. Außerdem sind diese JSON-Dateien sehr verbose und haben viel Overhead. Ein beispielhafter Ausschnitt ist in Code-Ausschnitt 4.6 zu sehen. Die eigentlichen Daten, die extrahiert werden sollen sind *Hero* mit *ID = 0* und *Size = 189.5*

```
  "@type": "g:Vertex",
  "@value": {
    "id": "0",
    "label": "Hero",
    "properties": {
      "Size": [
        {
          "@type": "g:VertexProperty",
          "@value": {
            "id": {
              "@type": "g:Int32",
              "@value": -361727145
            },
            "value": "189.5",
            "label": "Size"
          }
        }
```

```
18          ]
19        }
20    }
```

Code-Ausschnitt 4.6: Antwort-JSON von Neptune (Ausschnitt).

Zum Parsen dieser Struktur würde es sich anbieten die Go-Standardbibliothek zu verwenden, diese würde aber die gesamte Struktur parsen und die ganzen unnötigen Informationen mitverarbeiten, was unnötig viel Zeit beanspruchen würde. Für Proteus und Danos relevant sind aber nur das Vertex-Label *Hero*, sowie die Eigenschaften ID und Size. Um diese schnell und effizient zu extrahieren, wird ein custom Parser entworfen, der mit linearem Aufwand bezogen auf die Länge der JSON in Byte die relevanten Felder ausliest. Der gesamte in Go implementierte Algorithmus wäre an dieser Stelle zu umfangreich, wird also im Folgenden durch Algorithmus 1 als Pseudocode ausgedrückt. Die Funktion ist in der Lage eine beliebige Menge an Knoten zu parsen.

Algorithmus 1 Parse Vertex from Database

```
1: function ParseGSONVertex(gson)                    ▷ Graph JSON data as byte slice
2:     vertices ← empty vertex array
3:     while lengthOf(gson) > 0 do
4:         idIdx ← indexOf("id")
5:         if idIdx not found then
6:             break
7:         end if
8:         vertex ← isolate(gson[idIdx :])                  ▷ isolate vertex' raw data
9:         gson ← gson[endOfVertexIdx :]    ▷ deletes all data up until end of vertex
10:        id ← parseValue(vertex)                ▷ parse next value (id in this case)
11:        label ← parseField(vertex, "label")              ▷ get value of field "label"
12:        propIdx ← indexOf("properties")
13:        if propIdx not found then
14:            vertices.add(newVertexOf(id, label))          ▷ add new vertex to list
15:            continue
16:        end if
17:        props ← isolateProperties(vertex)
18:        p ← parsePropertiesVertex(props) ▷ loops over fields in properties and gets
   their names + values as map
19:        vertices.add(newVertexOf(id, label, p))
20:    end while
21:    return vertices
22: end function
```

4.3.2. Datenbankabfragen

Der Graphdatenbankadapter wird effektiv durch die Go-Struktur *Query*, die die Neptune-Endpoint-Adresse enthält, verkörpert. Alle Interaktionen mit der GDB erfolgen über die Query-Struktur.

Eine der wichtigsten Operationen ist das Erstellen neuer Knoten. Hierzu wird die Methode *AddVertex* (Code-Ausschnitt 4.7) verwendet. Sie nimmt einen Vektor v mit Struktur aus Abschnitt 4.3 entgegen, liest dessen Eigenschaften und fügt die ID hinzu, da diese

nach Unterabschnitt 2.2.3 wie eine Eigenschaft ohne Anführungszeichen als *Property* hinzugefügt werden muss. Dann wird die Gremlin-Abfrage zum Erstellen des Vektors mit dem Label von v erzeugt und danach die Hilfsfunktion *attachProperties* (Code-Ausschnitt 4.8) aufgerufen. Diese iteriert über die *properties*-Map und fügt Gremlin-Statements für jeden Key k und Value v hinzu. Hierbei ist der besondere Syntax für die ID zu beachten. Nachdem in *AddVertex* die Gremlin-Abfrage erstellt wurde, wird die ID wieder aus den Properties gelöscht, weil sie ein eigenes Feld hat und die Anfrage wird mit der Gremlin-Query per HTTPS an Neptune gesendet. Ein möglicher Fehler wird zurückgegeben.

```
1  func (q Query) AddVertex(v graph.Vertex) error {
2     m := v.Properties
3     if v.Properties == nil {
4        m = make(map[string]string)
5     }
6     m["Id"] = v.Id
7
8     gremlin := fmt.Sprintf("g.addV('%s')", label)
9     gremlin += attachProperties(properties)
10
11    delete(m, "Id")
12
13    _, err := sendRequest(q, gremlin)
14    return err
15 }
```

Code-Ausschnitt 4.7: Hinzufügen eines Knoten über *Query*-Datenbankadapter (angepasst)

```
1  func attachProperties(properties map[string]string) string {
2     gremlin := ""
3     for k, v := range properties {
4        if k == "Id" {
5           gremlin += fmt.Sprintf(".property(id,'%s')", v)
6           continue
7        }
8        gremlin += fmt.Sprintf(".property('%s','%s')", k, v)
```

```
 9       }
10       return gremlin
11   }
```

Code-Ausschnitt 4.8: Erstellen der Eigenschaftsliste von Knoten und Kanten für eine Gremlin-Abfrage

5. Fazit

In diesem finalen Kapitel wird nochmals das Vorgehen in dieser Arbeit zusammengefasst, eine kritische Reflexion vorgenommen, in der unter Anderem die Schwierigkeiten des Projekts aufgeführt werden, sowie eine Aussicht auf die Zukunft des Projekts gegeben. Abschließend wird eine Kosteneinschätzung für den Infrastruktur-Stack, der für Danos benötigt wird, durchgeführt. Dieser letzte Teil wird auch in englischer Sprache wiedergegeben, um die Einsicht für meine englischsprachigen Kollegen zu erleichtern.

5.1. Zusammenfassung

Hauptaufgabe war es, eine Plattform, ein Microservice zu schaffen, der es ermöglicht eine Recommendation Engine (RE) in eine Gamified Learning Platform (GLP) einzubinden. Diese Arbeit wurde zu Beginn mit einer Motivation eröffnet, um die Relevanz des Projektes, sowie dessen Ziele klar zu definieren. Daraufhin wurden die Grundlagentechnologien, die dieses Projekt überwiegend benötigte, um umgesetzt werden zu können, betrachtet. Hierauf konnte die Konzeptions- bzw. Planungsphase aufgebaut werden, die Erkenntnisse aus der Grundlagenbetrachtung nutzte, um fundierte Entscheidungen über die Architektur des Projektes treffen zu können. Eine dieser Dinge war die Erkenntnis, dass Graph Databases (GDBs) in den aller meisten Fällen geeigneter sind, als herkömmliche relationale Datenbanken. Die Konzeption umfasste außerdem eine Erörterung der außenstehenden Technologien, wie Proxys, um einen optimalen Einsatz zu planen und umsetzen zu können. Im letzten Teil - der Implementierung - wurde auf die Umsetzung der theoretischen Architektur und den theoretischen Vorgehensweisen aus der Konzeption erläutert. Hier wurden Schlüssel-Abschnitte des Codes gezeigt und erläutert.

5.2. Kritische Reflexion

Dieses Projekt arbeitet sehr stark mit extrem personenbezogenen Daten, was ein sehr hohes Niveau an Sicherheit verlangt. Hier wären die Verschlüsselungen im Transit und in der Ruhe zu nennen, die konzipiert und umgesetzt werden mussten. Auch musste sich ein Authentifizierungskonzept überlegt werden, welches einen sicheren Zugriff auf diese Daten erlaubt. Dieser Anspruch war schwieriger umzusetzen, da konstant sichergegangen werden musste, dass keine Sicherheitslücken bestehen. Dies ist vor allem wichtig für einen Service, der im öffentlichen Internet exponiert ist, was hier der Fall ist. Auch hat das Verwenden von AWS einige Sicherheitsaktionen nötig gemacht, die auf SAP-Hardware nicht notwendig gewesen wären, wie zum Beispiel das verschlüsseln der Datenbank.
Auch war es zu Beginn etwas schwierig sich in dem riesigen Dschungel von AWS-Services zurechtzufinden und das erbrachte Vertrauen, auf AWS Geld ausgeben zu können wurde zu einer kleinen Bürde, da auf AWS selbst keinerlei Preise stehen und immer in die Dokumentation geschaut werden musste. So hatte man beinahe das Gefühl mit jedem Klick Gefahr zu laufen ein kleines Vermögen auszugeben, und das auch noch ohne es zu Wissen. Außerdem war das Administrieren der vielen verwendeten Services und Technologien (AWS Route 53 für Domains, AWS Neptune, AWS Virtual Private Cloud (VPC), AWS EC2, AWS Controll Center, Nginx, Certbot, Let's-Encrypt, SSH, Amazon Linux) eine große Herausforderung, da sich in alle eingelernt werden musste, alle auf kommerzielle Nutzbarkeit überprüft werden mussten, alle auf Kosten überprüfte werden mussten, und das Administrieren von der eigentlichen Arbeit an dem Service abgehalten hat und immer wieder unterbrochen wurde. Nachdem allerdings alles erstmal aufgesetzt war, war es sehr einfach zu verwenden und auch das remotely Testen wurde sehr einfach, was nicht über die DHP gesagt werden kann, die in ihrem eigenen Dschungel von SAP-Systemen lebt, und nur sehr unflexibel geändert werden konnte. Mir wurde die Macht bewusst, die durch AWS zur Verfügung steht und wie damit hoch skalierbare Systeme gebaut werden können.
Ein weiterer der schwierigeren Punkte war es, eine Struktur der Graph-Daten zu entwerfen, die flexibel genug ist, um sich an die noch nicht ausreichend definierten Daten der Heroes und Mission anzupassen, aber auch nicht zu generisch ist, um einen Datenbankadapter beinahe unnötig zu machen.
Eine leichtere Übung hingegen war das Implementieren einer guten REST API, da es hierfür extrem viele gute Ressourcen zu finden gibt und Go standardmäßig dies sehr gut unterstützt und als Backend-Sprache unter anderem auch dafür konzipiert wurde.

Die Danos-Software wird derzeit (Stand: 14. Dez. 2020) von Seiten der DHP, sowie der Danos-Engine noch implementiert und soll Mitte Januar 2021 vollendet werden, um eine erste Beta-Version der Danos-Engine auf den Production-Servern der DHP bereitzustellen. Damit ist diese Arbeit eine wichtige Vorbereitung für diese Integration. Für die Danos-Engine müssen noch ein paar wenige Anpassungen an *Proteus* vorgenommen werden, damit die Implementierung des Caches auf die von Danos definierten Datenstrukturen passt. Danach sollten Mitte Januar nur noch die Steine zusammengesteckt werden müssen. Sollte die Implementierung erfolgreich sein und Danos halten, was es verspricht, kann *Proteus* ohne weitere Modifikation über viele Instanzen eingesetzt werden, um viele DHP-Instanzen mit tausenden von Heroes gleichzeitig zu bedienen.

5.3. Kostenbetrachtung

Dieser Abschnitt wird die Kosten für *Proteus*-Instanzen auf AWS in Abhängigkeit von der Zahl an DHP-Instanzen untersuchen. Dafür muss ermittelt werden, wie viele AWS-Instanzen und vor allem von welchem Typ benötigt werden, und wie kostspielig diese sind. Alle Währungen sind in US-Dollar angegeben.

Neptune-Instanzen:
Jede DHP-Instanz die ihre eigene Hana-DB-Instanz hat, benötigt auch eine eigene Neptune-DB-Instanz. Außerdem wird eine weitere Neptune-Instanz für die Entwicklung benötigt. Der kleinste Typ hat 2 vCPUs, 4 GB RAM, und 1500 Mb/s Bandbreite bei einem Preis von $0,115 pro Stunde ($84,18 monatlich). Diese Instanz sollte in der Lage sein 3 000 bis 5 000 Heroes zu bedienen. Die nächst größere Instanz wäre bei $0,42 pro Stunde ($307 monatlich). Zusätzlich kosten Traffic und IO-Operationen extra. Für die kleinere Instanz und für weniger als 3 000 Heroes nehme ich extra Kosten von ca. $40 pro Monat an.

EC2-Container:
Von einem *Proteus*-Standpunkt sollte eine Instanz genügen, um mehrere tausend Heroes zu bedienen. Allerdings ist Danos in meinem Verständnis sehr ressourcen-intensiv und fordernd, was bedeutet, dass es wahrscheinlich nicht mehrere tausend Heroes mit einer Instanz bedienen kann. Die Instanz sollte auch relativ mächtig sein und einige Kerne haben, um Danos Parallelismus auszunützen. Die kleinste Instanz die ich für ein Production-

Umfeld als geeignet erachte liegt preislich bei $0,1164 pro Stunde ($85,2 monatlich) und hat 4 vCPUs und 8 GiB RAM. Diese Instanz könnte in der Lage sein bis zu 1 000 oder 3 000 Heroes zu unterstützen. Jedoch ist das nicht sicher, da ich eigentlich keine Ahnung habe, wie gut bzw. schlecht Danos sich ressourcen-technisch verhält. Wie auch immer, in allen Fällen kommt noch eine Instanz für die Entwicklung hinzu. Diese kostet $0,0291 pro Stunde, also $21,3 monatlich.

Berechnung:
Sei n die Zahl an DHP-Instanzen (tenants) in der jeweiligen Hero-Anzahl-Kategorie. Dann zeigt Tabelle 5.1 einige Formeln zur Berechnung der Kosten von Danos bei n DHP-Instanzen. Sollten die DHP-Instanzen in unterschiedlichen Hero-Kategorien sein, so muss man die Teilergebnisse addieren. Es sollte auch gesagt sein, dass für die Entwicklungsumgebung monatliche Kosten von ca. $90 dazukommen.

Heroes	**Optimistisch**	**Pessimistisch**
≤ 3 000	$\$(124,18 + 85,2)n$	$\$(124,18 + 3 \cdot 85,2)n$
> 3 000	$\$(124,18 + 5 \cdot 85,2)n$	$\$(367 + 8 \cdot 85,2)n$

Tabelle 5.1.: Formeln zu Berechnung der monatlichen Kosten; mit n der Anzahl an DHP-Instanzen.

Rechenbeispiel:
Sei n_1 gleich zwei DHP-Instanzen mit weniger als 3 000 Heroes und n_2 eine DHP-Instanz mit mehr als 3 000 Heroes. Inklusive der Kosten für die Entwicklungsumgebung sind die monatlichen Kosten:

Optimistisch: $\$(124,18 + 85,2) \cdot 2 + \$(124,18 + 5 \cdot 85,2) + \$90 = \$1058,94$
Pessimistisch: $\$(124,18 + 3 \cdot 85,2) \cdot 2 + \$(367 + 8 \cdot 85,2) + \$90 = \$1898,16$

Literaturverzeichnis

[1] Stephan Kamps. „How to Become a Digital Hero (No Cape Required)“. In: *SAP News Employee Network* (2018-09-21). URL: https://blogs.wdf.sap.corp/sapnews_en/2018/09/how-to-become-a-digital-hero-no-cape-required/ (Einsichtnahme: 08. 12. 2020).

[2] Kabat, M. „Vorreiter für digitale Innovationen“. In: *SAP News Germany* (2019-02-04). URL: https://news.sap.com/germany/2019/02/florian-roth-cio-interview/ (Einsichtnahme: 08. 12. 2020).

[3] Florian Roth. *Florian Roth talking about the Digital Heroes Initiative.* 2020-12-08. URL: https://broadcast.co.sap.com/vo/portal/191205_DigHeroes_v03 (Einsichtnahme: 08. 12. 2020).

[4] James Hoskin. *Architecture.* 2020-06-26. URL: https://github.wdf.sap.corp/digitalheroes/octopus-dh-backend/wiki/1-:-Architecture (Einsichtnahme: 08. 12. 2020).

[5] Germanakos, P./ Defiebre, D. *A Human-centered Business Scenario in SIoT - The Case of DANOS Framework.*

[6] Ricci, F./ Rokach, L./ Shapira, B. „Introduction to Recommender Systems Handbook“. In: *Recommender systems handbook.* Hrsg. von Ricci, F. New York: Springer, 2011, S. 1–35. URL: http://www.inf.unibz.it/~ricci/papers/intro-rec-sys-handbook.pdf (Einsichtnahme: 08. 12. 2020).

[7] Germanakos, P./ Sacharidis, D./ Defiebre, D. *A Decentralized Recommendation Engine in the Social Internet of Things.*

[8] *Danos - Dynamic and Anthropomorphic Network of Objects Simulator.* 2020-12-08. URL: https://experience-garage.int.sap/en/custom/accelerator/view/346 (Einsichtnahme: 08. 12. 2020).

[9] SAP CreateX Innovation Platform. *SAP CreateX Innovation Platform.* 2020-12-08. URL: https://experience-garage.int.sap/en/page/employee-ideas-en/ (Einsichtnahme: 08. 12. 2020).

[10] Amazon Web Services, Inc. *Amazon Web Services AWS – Server Hosting & Cloud Services*. 2020-12-07. URL: https://aws.amazon.com/de/ (Einsichtnahme: 08. 12. 2020).

[11] *Frequently Asked Questions (FAQ) - The Go Programming Language*. 2020-12-08. URL: https://golang.org/doc/faq#go_or_golang (Einsichtnahme: 08. 12. 2020).

[12] *The Go Programming Language Specification - The Go Programming Language*. 2020-12-08. URL: https://golang.org/ref/spec#Introduction (Einsichtnahme: 08. 12. 2020).

[13] *The Go Programming Language*. 2020-12-08. URL: https://golang.org/ (Einsichtnahme: 08. 12. 2020).

[14] Kincaid, J. „Google's Go: A New Programming Language That's Python Meets C++". In: *TechCrunch* (2009-11-11). URL: https://techcrunch.com/2009/11/10/google-go-language/ (Einsichtnahme: 08. 12. 2020).

[15] *Frequently Asked Questions (FAQ) - The Go Programming Language*. 2020-12-08. URL: https://golang.org/doc/faq#history (Einsichtnahme: 08. 12. 2020).

[16] *The Go Project - The Go Programming Language*. 2020-12-08. URL: https://golang.org/project/#go1 (Einsichtnahme: 08. 12. 2020).

[17] *Google TechTalks October 30, 2009*. 2009-10-30. URL: https://www.youtube.com/watch?v=rKnDgT73v8s (Einsichtnahme: 08. 12. 2020).

[18] *Go at Google: Language Design in the Service of Software Engineering*. 2012-10-25. URL: https://talks.golang.org/2012/splash.article (Einsichtnahme: 08. 12. 2020).

[19] *Stack Overflow - Where Developers Learn, Share, & Build Careers*. 2020-12-08. URL: https://stackoverflow.com/ (Einsichtnahme: 08. 12. 2020).

[20] Stack Overflow. *Stack Overflow Developer Survey 2020*. 2020. URL: https://insights.stackoverflow.com/survey/2020 (Einsichtnahme: 08. 12. 2020).

[21] Github, Inc. *The State of the Octoverse*. 2020-12-03. URL: https://octoverse.github.com/ (Einsichtnahme: 08. 12. 2020).

[22] *Git - git-request-pull Documentation*. URL: https://git-scm.com/docs/git-request-pull (Einsichtnahme: 08. 12. 2020).

[23] *Github Language Stats*. 2020-10-03. URL: https://madnight.github.io/githut/#/pull_requests/2020/3 (Einsichtnahme: 08. 12. 2020).

[24] Doxsey, C. *Introducing Go: Build reliable, scalable programs / Caleb Doxsey.* First edition. Beijing: O'Reilly, 2016. URL: https://www.golang-book.com/books/intro (Einsichtnahme: 09. 12. 2020).

[25] *Frequently Asked Questions (FAQ) - The Go Programming Language.* 2020-12-09. URL: https://golang.org/doc/faq#principles (Einsichtnahme: 09. 12. 2020).

[26] Bjarne Stroustrup. *Stroustrup: C++ Glossary.* 2020-11-23. URL: https://www.stroustrup.com/glossary.html#Gpolymorphism (Einsichtnahme: 09. 12. 2020).

[27] *Graph DBMS - DB-Engines Encyclopedia.* 2020-12-09. URL: https://db-engines.com/en/article/Graph+DBMS (Einsichtnahme: 09. 12. 2020).

[28] *Amazon Neptune System Properties.* 2020-12-09. URL: https://db-engines.com/en/system/Amazon+Neptune (Einsichtnahme: 09. 12. 2020).

[29] *Most Popular Graph Databases.* 2019-07-10. URL: https://www.c-sharpcorner.com/article/most-popular-graph-databases/ (Einsichtnahme: 09. 12. 2020).

[30] *When (and Why) to Choose Graph Databases over Relational Databases.* 2019-08-02. URL: https://leapgraph.com/graph-vs-relational-databases (Einsichtnahme: 09. 12. 2020).

[31] Litzel, N. „Was ist eine Graphdatenbank?" In: *BigData-Insider* (2019-10-01). URL: https://www.bigdata-insider.de/was-ist-eine-graphdatenbank-a-788834/ (Einsichtnahme: 09. 12. 2020).

[32] *DB-Engines Ranking per database model category.* 2020-12-09. URL: https://db-engines.com/en/ranking_categories (Einsichtnahme: 09. 12. 2020).

[33] *Querying a Neptune Graph - Amazon Neptune.* 2020-12-07. URL: https://docs.aws.amazon.com/neptune/latest/userguide/access-graph-queries.html (Einsichtnahme: 09. 12. 2020).

[34] TinkerPop, A. *Apache TinkerPop.* 2020-11-09. URL: https://tinkerpop.apache.org/ (Einsichtnahme: 09. 12. 2020).

[35] *The Benefits of the Gremlin Graph Traversal Machine | Datastax.* 2020-12-09. URL: https://www.datastax.com/blog/benefits-gremlin-graph-traversal-machine (Einsichtnahme: 09. 12. 2020).

[36] Lawrence, K. R. *PRACTICAL GREMLIN: An Apache TinkerPop Tutorial*. 2020-10-11. URL: https://kelvinlawrence.net/book/Gremlin-Graph-Guide.html (Einsichtnahme: 09. 12. 2020).

[37] *TinkerPop Documentation*. 2020-08-03. URL: https://tinkerpop.apache.org/docs/current/reference/ (Einsichtnahme: 09. 12. 2020).

[38] *Gremlin Standards Compliance in Amazon Neptune - Amazon Neptune*. 2020-12-07. URL: https://docs.aws.amazon.com/neptune/latest/userguide/access-graph-gremlin-differences.html (Einsichtnahme: 09. 12. 2020).

[39] Richter, F. „Amazon Leads $100 Billion Cloud Market“. In: *Statista* (2019-26-07). URL: https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/ (Einsichtnahme: 09. 12. 2020).

[40] Amazon Web Services, Inc. *Amazon EC2*. 2020-11-09. URL: https://aws.amazon.com/ec2/?ec2-whats-new.sort-by=item.additionalFields.postDateTime&ec2-whats-new.sort-order=desc (Einsichtnahme: 09. 12. 2020).

[41] *Supported operating systems - AWS Systems Manager*. 2020-12-07. URL: https://docs.aws.amazon.com/systems-manager/latest/userguide/prereqs-operating-systems.html (Einsichtnahme: 09. 12. 2020).

[42] Amazon Web Services, Inc. *Amazon Neptune - Fast, Reliable Graph Database built for the cloud*. 2020-12-08. URL: https://aws.amazon.com/neptune/ (Einsichtnahme: 09. 12. 2020).

[43] *Accessing the Neptune Graph with Gremlin - Amazon Neptune*. 2020-12-07. URL: https://docs.aws.amazon.com/neptune/latest/userguide/access-graph-gremlin.html (Einsichtnahme: 09. 12. 2020).

[44] *Encrypting Neptune Resources at Rest - Amazon Neptune*. 2020-12-07. URL: https://docs.aws.amazon.com/neptune/latest/userguide/encrypt.html (Einsichtnahme: 09. 12. 2020).

[45] *Encryption in Transit: Connecting to Neptune Using SSL/HTTPS - Amazon Neptune*. 2020-12-07. URL: https://docs.aws.amazon.com/neptune/latest/userguide/security-ssl.html (Einsichtnahme: 09. 12. 2020).

[46] *Overview of Backing Up and Restoring a Neptune DB Cluster - Amazon Neptune*. 2020-12-07. URL: https://docs.aws.amazon.com/neptune/latest/userguide/backup-restore-overview.html (Einsichtnahme: 09. 12. 2020).

[47] Scott S. Sheppard. *NeptuneMoons*. 2020-12-10. URL: https://sites.google.com/carnegiescience.edu/sheppard/moons/neptunemoons (Einsichtnahme: 10. 12. 2020).

[48] Kerényi, K. *The Gods of The Greeks*. Pickle Partners Publishing, 2016.

[49] *gRPC*. 2020-12-11. URL: https://grpc.io/ (Einsichtnahme: 11. 12. 2020).

[50] Baeldung. *JAX-RS Client with Jersey*. 2017. URL: https://www.baeldung.com/jersey-jax-rs-client (Einsichtnahme: 11. 12. 2020).

[51] *What is an SSL Certificate? | How to Get a Free SSL Certificate*. 2020-12-11. URL: https://www.cloudflare.com/learning/ssl/what-is-an-ssl-certificate/ (Einsichtnahme: 11. 12. 2020).

[52] *Working with Certificates and SSL (Sun Java System Application Server Platform Edition 8.2 Administration Guide)*. 2011-01-27. URL: https://docs.oracle.com/cd/E19830-01/819-4712/ablqw/index.html (Einsichtnahme: 11. 12. 2020).

[53] *Let's Encrypt - Free SSL/TLS Certificates*. 2020-12-11. URL: https://letsencrypt.org/ (Einsichtnahme: 11. 12. 2020).

[54] *Certbot*. 2020-12-08. URL: https://certbot.eff.org/ (Einsichtnahme: 11. 12. 2020).

[55] *Data protection in Amazon Virtual Private Cloud - Amazon Virtual Private Cloud*. 2020-12-09. URL: https://docs.aws.amazon.com/vpc/latest/userguide/data-protection.html (Einsichtnahme: 11. 12. 2020).

[56] *What is Amazon VPC? - Amazon Virtual Private Cloud*. 2020-12-09. URL: https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html (Einsichtnahme: 11. 12. 2020).

[57] *nginx*. 2020-12-02. URL: https://nginx.org/en/ (Einsichtnahme: 11. 12. 2020).

[58] „November 2020 Web Server Survey“. In: (2020-11-30). URL: https://news.netcraft.com/archives/2020/11/30/november-2020-web-server-survey.html (Einsichtnahme: 11. 12. 2020).

[59] *Authorization*. 2020-12-14. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Authorization (Einsichtnahme: 14. 12. 2020).

[60] *AWS Shield*. 2020-11-09. URL: https://aws.amazon.com/shield/?whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=desc (Einsichtnahme: 15. 12. 2020).

A. Anhang

A.1. Evaluation of Expenditures (English Version of *5.3 Kostenbetrachtung*)

This is the English version of *5.3 Kostenbetrachtung* as reference for my international colleagues.[1]

This section will scrutinize the expenditures of sustaining *Proteus*[2] instances on Amazon Web Services (AWS) in dependence of the number of Digital Hero tenants. To this end, it needs to be determined how many AWS instances of which type are required and how costly each of them are. All currencies are in US Dollar.

Neptune Instances:
Each Digital Heroes Platform (DHP) instance that has its own Hana Database (DB) instance will also require its own Neptune DB instance. Furthermore, one additional Neptune instance will be required for development. The smallest type features 2 vCPUs, 4 GB RAM, and 1500 Mbps throughput at a price of $0.115 per hour ($84.18 monthly). This instance should be able to sustain up to 3 000 to 5 000 Heroes. The next more powerful instance would be at $0.42 per hour ($307 montly). Additionally, traffic and IO-ops cost extra. For the smaller instance and for less than 3 000 Heroes I assume at most an extra $40 per month.

EC2 Container:
From a Proteus standpoint, one instance would be sufficient to serve several thousand Heroes. However, to my understanding, Danos is very resource-intensive and demanding, meaning that it probably cannot sustain several thousand Heroes with just one instance. The instances will also need to be rather powerful with several cores in order to make use of Danos' multi-threading. The smallest instance I reckon feasible for a production environment is coming in at $0.1164 per hour ($85.2 monthly), featuring 4 vCPUs and 8

[1] I only realized that I could've written this entire paper in English when I was already halfway done with it in German.
[2] *Proteus* is the name given to the AWS-sided implementation of Danos. Danos runs within Proteus.

GiB RAM. This instance may be able to support up to 1 000 to 3 000 Heroes. However, this is not certain as I have no knowledge of how poorly/well Danos performs resource-wise. Either way, one additional instance will be required for development, coming in at $0.0291 per hour ($21.3 monthly).

Calculation:
Let n be the number of DHP instances (tenants) in their respective Hero count category. Then table A.1 provides several formulas for calculating the costs of supplying DHP instances with Danos. If the instances are in different Hero categories, just add the results. Remember that for the development environment there is an additional monthly cost of about $90.

Heroes	**Optimistic**	**Pessimistic**
≤ 3 000	$\$(124.18 + 85.2)n$	$\$(124.18 + 3 \cdot 85.2)n$
> 3 000	$\$(124.18 + 5 \cdot 85.2)n$	$\$(367 + 8 \cdot 85.2)n$

Tabelle A.1.: Montly Expenditure Formulas; n being the number of DHP instances.

Example:
Let n_1 be two tenants with less than 3 000 Heroes and n_2 a tenant with greater than 3 000 Heroes. Plus the expenditures for the development environment, the monthly costs are:

Optimistic: $\$(124.18 + 85.2) \cdot 2 + \$(124.18 + 5 \cdot 85.2) + \$90 = \$1058.94$
Pessimistic: $\$(124.18 + 3 \cdot 85.2) \cdot 2 + \$(367 + 8 \cdot 85.2) + \$90 = \$1898.16$

A.2. Dokumentation

Das nachfolgende Dokument entspricht der Dokumentation von Proteus zum 16. Dezember 2020. Die aktuellste Version ist stets auf https://github.wdf.sap.corp/danos/proteus zu finden.

Proteus

Proteus is the name of an infrastructure API used by the Danos engine to provide recommendations to the Digital Heroes Platform. The internal package used by the Digital Heroes backend to communicate with this API's REST API is also called Proteus.

The current domain for accessing this REST API as per this security concept is `sap-proteus.de`.

Table of Contents

- Usage
  - Initializing the API
- Architecture
  - Package Dependencies
- Deployment and Administration
  - REST API Access Permissions
  - Testing
    - Gremlin DB
    - REST API
- REST API
  - Authorization
  - Endpoints
    - General HTTP Error Codes
  - General Rules for JSON
  - GET
  - POST
    - Missions and Heroes
    - Ratings
  - PUT
    - Missions and Heroes
    - Ratings
  - DELETE
- Attribution

Usage

As a user who wants to implement the Danos engine using this infrastructure API, you need to do the following things:

1. Call `proteus.Initialize(...)` to setup your environment.
2. Now you may start the REST API - which is listening for external updates - with `proteus.StartRestApi(...)`. It opens in a separate Goroutine.
3. At last, you may get the API interface with `proteus.GetCache()`, which you will use for all of your interactions with this API.

It is not advised to access the public methods/functions of the "sub"-packages of proteus. You may do so to write your own, custom gremlin queries, though. However, be aware doing so might give you inconsistent data from the API interface. Proceed with caution and only if you know what you are doing.

Initializing the API

To initialize the API via a call to `proteus.Initialize(...)` you will need to provide a few arguments:

- permissionFilePath - The file system path to the REST API access permissions file (see REST API Access Permissions). This path will need to be provided to you by your system administrator.
- neptuneEndpoint - The http endpoint of your Neptune instance. This will be given to you by your system administrator. More information on that may be found here.
- callback - Here you will need to provide a function that has a string as parameter and returns a string slice along with an error. This function will be called upon accessing GET /api/recommendations/{hid}. Your function will be given a Hero ID and should return an ordered list of recommended Mission IDs. The first Mission ID should be the most recommended, followed by the second most recommended and so on. You do not need to check whether the Hero exists; if your function gets called, it does. This function is your hook.

As a general rule of thumb: Always also look at the Go-Documentation directly taken from the code.

The initialization can fail for a number of reasons: - DB connection not established - DB cannot be read from - Permission file not found or not parsed properly

If the function doesn't panic, you will be all set to use the other methods of the `proteus` API in any order you like.

When you call `proteus.StartRestApi(...)`, you will need to provide an address/port to which the web server should listen. This port will be provided to you by your system administrator. The default is `:8080`. More information may be found here.

Architecture

Package Dependencies

Proteus-Dependencies

The above picture shows how the subpackages are interdependent. It also shows the touching points with the outside infrastructure.

Deployment and Administration

Proteus runs on a single EC2 instance that has access to the AWS Neptune DB.

The REST API listens for HTTP on port `:8080` and is provided with connections by the NGINX reverse proxy. NGINX is only available over HTTPS, so that the communication is encrypted end-to-end. For the SSL certificates, NGINX uses certbot with Let's Encrypt certificates. The currently used NGINX config is found here. Be sure to set up auto-renewal with certbot and perform a dry-run to test it. Never route unencrypted traffic over a network!

Access to the REST API is managed with a permissions.json file, usually located under `/etc/proteus/permissions.json`.

REST API Access Permissions

Permissions and API Access (via API token) are managed with the following JSON file, usually located under `/etc/proteus/permissions.json` on the EC2 instance:

```
{
  "keys": [
    {
      "id": "userid",
      "secret": "SOME_SHA1-HASHED_HEX-ENCODED_SECRET",
      "permission": ["", "extended_permission_1"]
    },
    {
      "id": "useridbasic",
      "secret": "SOME_OTHER_SHA1-HASHED_HEX-ENCODED_SECRET",
      "permission": [""]
    }
  ]
}
```

The secret has to be SHA1-hashed and HEX encoded in this file. To create a new API token generate a random and secure password (with numbers, lower-case and upper-case letters, special characters and length 50), hash it, encode it and put it in the file as 'secret' to your custom-defined user-'id'. Give the plain-text secret to your trusted user over a secure channel.

To get an idea of the hashing and encoding you may use this website. However, note that for production API tokens, the secrets should be hashed and encoded manually, or with approved software, for security reasons.

To give basic permissions to a user/API token add an empty string (`""`) to the permission array of the token. Some operations, however, will require extended permissions. Add those to an API token where needed. A key / token has only the permissions that are explicitly listed in their permission array.

Testing

The tests that can be executed locally are placed with the files containing the functions/methods as *_test.go.

Tests that can only be executed properly on the EC2 instance are in their own separate *_test directories. For these there are scripts for convenience:

Gremlin DB

To test your database setup and the Neptune/Gremlin DB queries have a look at the gremlin_test.sh script. Place your danos.ppk SSH private key to the EC2 instance in the directory above this one and adjust your EC2 endpoint. (As of now the dev endpoint is `ec2-user@ec2-18-159-0-151.eu-central-1.compute.amazonaws.com:/home/ec2-user/gremlin.test` and will not change, unless the AWS elastic IP is revoked or changed).

If the Neptune DB endpoint changes, you have to change the constant in the gremlin/gremlin_test/gremlin_test.go file.

REST API

To spin up the production-like REST API, use the proteus_test.sh script which is constructed similarly to the gremlin_test.sh. Proceed with the configuration as in testing Gremlin DB.

If the Neptune DB endpoint changes, you have to change the constant in the proteus_test/proteus_test.go file.

Note that this REST API will write to the DB and works exactly like the one used in production. Use it for testing with Insomnia/Postman or the DH-Backend.

REST API

Hero IDs and Mission IDs should have their separate ID spaces. They should be **unique**.

The current development address is `sap-proteus.de`. Note that you should **always** access the resources over HTTPS. Do NOT send unencrypted headers with `Basic` authentication.

Authorization

If you work with this API, you should have been provided with an API token in the form of a user ID, a secret and a list of permissions. If not, contact an admin.

With the user ID and secret you will gain access to the API endpoints if you provide them with the 'Basic' method in the 'Authentication' header field. You need to provide the credentials with every request.

The endpoints are HTTP**S** only. This API uses SSL certificates issued by Let's encrypt. If they are not trusted by your environment, go ahead and install them.

```
! NEVER TRY TO ACCESS WITHOUT TLS !
```

Endpoints

All response bodies will be of `Content-Type: application/json`. Note that `hid` and `mid` stand for a specific hero and mission id, respectively.

The following endpoints currently exist:

- GET (Request: No Body; Permissions: Basic)
  - `/api/recommendations/{hid}`
- POST (Request: `Content-Type: application/json`; Permissions: Basic)
  - `/api/heroes`
  - `/api/missions`
  - `/api/ratings`
- PUT (Request: `Content-Type: application/json`; Permissions: Basic)
  - `/api/heroes/{hid}`
  - `/api/missions/{mid}`
  - `/api/ratings/{hid}/{mid}`
- DELETE (Request: No Body; Permissions: Basic)
  - `/api/heroes/{hid}`
  - `/api/missions/{mid}`
  - `/api/ratings/{hid}/{mid}`

General HTTP Error Codes

▶ The following general errors may be returned for every request. They are ordered by their probable occurrence:

General Rules for JSON

- All fields must start with a letter.
- There must not be any two fields with the exact same name except for case-sensitivity of their first letter (e.g. no fields 'myField' and 'MyField').
- All values (except `properties` itself) must be of type `string` (naturally, you can still put numbers in those strings).
- The `properties` field may have any number of fields. Ratings must have at least `rating`.
- Other top-level fields that are not required by an API-endpoint will be ignored.

GET

GET will return `200 OK` upon success and `404 Not Found` if the provided Hero ID (`hid`) doesn't exist.

The response body will be a JSON array of recommended Mission IDs (`mid`) ordered by the quality of the recommendation (i.e. they are ordered from the best to the worst recommendation) like the following:

```
[
  "mid0",
  "mid1",

  "midx"
]
```

POST

POST requests must always specify a body of `Content-Type: application/json`.

POST requests will return `201 - Created` upon success, and `409 - Conflict` if the specified ID already exists.

Missions and Heroes

The JSON for this request must include an `id` and `properties` field:

```
{
  "id": "uid",
  "properties": {
    "property0": "value0",
    "property1": "value1",

    "propertyX": "valueX"
  }
}
```

Ratings

The JSON for this request must include an `hid`, `mid`, `properties`, and `properties/rating` field:

```
{
  "hid": "uid",
  "mid": "uid",
  "properties": {
    "rating": "13.37",

    "additionalProperty0": "value0",
    "additionalProperty1": "value1",
    "additionalPropertyX": "valueX"
  }
}
```

PUT

PUT requests must always specify a body of `Content-Type: application/json`. If you create a PUT request, all properties of the modified entity must be repeated, or they will be deleted.

PUT requests will return `404 - Not Found` if the requested resource does not exist, it will **not** create the resource. Upon successfully updating, PUT requests return `204 - No Content`.

Missions and Heroes

The JSON for this request must include a `properties` field:

```
{
  "properties": {
    "property0": "value0",
    "property1": "value1",

    "propertyX": "valueX"
  }
}
```

Ratings

The JSON for this request must include a `properties`, and `properties/rating` field. You must include `rating`, it cannot be deleted:

```
{
  "properties": {
    "rating": "13.37",

    "additionalProperty0": "value0",
    "additionalProperty1": "value1",
    "additionalPropertyX": "valueX"
  }
}
```

DELETE

Upon successfully deleting, DELETE requests return `204 - No Content`. They return `404 - Not Found` if the specified entity does not exist.

Please note that upon deleting a Hero or a Mission, any Edges connected will be **automatically deleted** as well. You do not have to do this manually.

Attribution

The package proteus/gatekeeper uses github.com/gorilla/mux v1.8.0. The authors of proteus and proteus/gatekeeper comply with gorilla-mux' license by disclosing this very license in the documentation (this README) of this software API.