TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

Diplomarbeit

Gestaltung einer API zur Datenerfassung und -verwaltung mittels Eve

Forschungsbereich MaschinenbauInformatik und Virtuelle Produktentwicklung
Univ.-Prof. Dr.-Ing. Detlef Gerhard

von

Oliver Willem
0826532
Hintzerstraße 5/9
1030 Wien

Wien, am 18.06.2017 .....

Eidesstattliche Erklärung

Ich habe zur Kenntnis genommen, dass ich zur Drucklegung meiner Arbeit unter der Bezeichnung

Diplomarbeit

nur mit Bewilligung der Prüfungskommission berechtigt bin.

Ich erkläre an Eides statt, dass ich meine Diplomarbeit nach den anerkannten Grundsätzen für wissenschaftliche Abhandlungen selbständig ausgeführt habe und alle verwendeten Hilfsmittel, insbesondere die zugrunde gelegte Literatur genannt habe.

Wien, am 18.06.2017 .....

Kurzfassung

Mit steigendem Grad der Technisierung und voranschreitenden Entwicklungen im Gebiet der Industrie 4.0 wird das Erfassen und Verwalten von Daten ein immer größerer Aufgabenbereich. Ein Teilgebiet davon ist das Management von Daten, die zum Testen und Überwachen von Maschinen nötig sind.

Für deren Einführung wird meist großes Fachwissen im Bereich der Informationstechnik benötigt. Wenn in einem Betrieb kein Informatiker vorhanden ist, muss entweder einer von außen hinzugezogen werden oder jemand aus einem anderen Fachbereich muss sich, eventuell von Grund auf, mit diesem Thema auseinandersetzen.

In dieser Arbeit soll ein System geschaffen werden, das hilft dieses Problem besser zu bewältigen. Auch Nutzer mit geringem Informationstechnikwissen sollen die Möglichkeit haben, ein individuelles Datenerfassungssystem zu implementieren. Ein wichtiger Punkt ist dabei die einfache und übersichtliche Gestaltung.

Dazu wird in dieser Arbeit ein Webservice, genauer ein Representational State Transfer (REST) Application Programming Interface (API), geschaffen. Diese API ist gut skalierbar. Auch nachdem sie beispielsweise für Testzwecke implementiert wurde, kann sie in ein laufendes System eines Betriebs eingefügt werden. Neben dem Speichern und Verwalten von Daten in einer MySQL-Datenbank ermöglicht sie die Steuerung von Sensorsystemen. Die API kann die gewonnenen Daten in einem Netzwerk bereitstellen und verfügt über einen Authentifikationsmechanismus sowie Rollen- und Rechtevergabe, um diese Daten anschließend zu schützen.

Im ersten Teil der Arbeit geht es um die grundlegende Theorie und die Prozesse der Entscheidungsfindung zugunsten bestimmter Komponenten. Im zweiten Teil wird zunächst der Aufbau der API erklärt. Danach wird die Handhabe der Anwendung erläutert. Es folgt die Beschreibung einer konkreten Verwendungsmöglichkeit: Das Konzept beinhaltet die Installation der API auf einem günstigen Einplatinencomputer (Raspberry Pi), mit dessen Hilfe gezeigt wird, wie die API Sensoren steuern und die ausgelesenen Daten verwalten kann.

Ergebnis ist ein Leitfaden zur schnellen, günstigen und einfachen Anwendung der API. Das vorgestellte System ermöglicht die rasche Erfassung und Verwaltung von Daten in Testfällen, hohe Skalierbarkeit, dauerhaften Betrieb und Anbindung an bestehende Systeme. Die Implementierung ist so gestaltet, dass sie auch ein unerfahrener Nutzer durchführen kann.

Abstract

With the increasing degree of technization and the development in sectors like Industry 4.0, the collection and maintenance of data is becoming an increasingly important task. Part of this is the management of data that is needed to test and monitor machines.

The implantation of such a data management system requires a great deal of knowledge in the field of information technology. In mechanical engineering operations, however, there is often no one with the appropriate knowledge. Then a computer scientist has to be drawn from the outside or someone from another field has to acquire the abilities needed for this work, which can take quite some time.

In this thesis, a system will be created that helps to accomplish this task by allowing the user to implement an individual data acquisition system with little information technology knowledge. An important point is the simple and clear design.

A web service, more precisely a Representational State Transfer (REST) Application Programming Interface (API), is created to deal with this problem. This API is not only scalable, but also easy to be implemented into an existing system after using it for test purposes. In addition to storing and managing data in a database, it enables also the control of sensor systems used for data collection. The API can provide the data in a network and has an authentication mechanism as well as roles for users to secure data.

The first part of the work deals with the basic theory and the processes of decision-making in favor of using certain components over others. The second part explains the structure of the API. Then the handling of the application itself is shown. A description of a specific usage of the API follows: The concept involves installing the API on a low-cost single-board computer (Raspberry Pi), showing how the API can control sensors and manage the so gathered data.

Thus a concept for a fast, inexpensive and simple application can be presented. It enables to quickly capture and manage data in test cases, supports high scalability, permanent operation and connection to existing systems. The implementation is designed in such a way that it can also be carried out by an inexperienced user.

Inhaltsverzeichnis

Abkürzungsverzeichnis

AOP	Aspektorientierte Programmierung
API	Application Programming Interface
CA	Certificate Authority
DRY	Don't Repeat Yourself
HATEOAS	Hypermedia As The Engine Of Application State
HTTP	HyperText Transfer Protocol
IT	Informationstechnik
JSON	JavaScript Object Notation
MPM	Multi-Processing Modules
OOP	Objektorientierten Programmierung
OS	Operating System
PEP	Python Enhancement Proposal
REST	Representational State Transfer
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SSL	Secure Socket Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDDI	Universal Description, Discovery, and Integration
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VM	Virtuelle Maschine
WAF	Web Application Framework
WSD	Web Service Description
WSDL	Web Service Description Language
WSGI	Web Server Gateway Interface
XML	Extensible Markup Language

1 Einleitung

1.1 Problemstellung

Im Zuge der vierten industriellen Revolution ist die Verarbeitung von Daten ein Aufgabenbereich, der ständig an Bedeutung gewinnt. Die Auswertung von Sensordaten erfordert deren gewissenhafte Speicherung und anschließende Bereitstellung. Dazu existiert eine Vielzahl von Möglichkeiten. Deren Implementierungen benötigen jedoch meist großes Fachwissen.

Ein Informatiker, der die gewünschten Fähigkeiten besitzt, um diese Aufgabe zu bewältigen, ist in Betrieben des Fachbereichs Maschinenbau jedoch nicht immer vorhanden oder verfügbar. Gerade in kleineren Unternehmen ist dadurch die Umsetzung einer solchen Datenerfassung durch einen Informatiker nur sehr schwer oder gar nicht möglich.

Einen vorhandenen Experten aus dem Fachbereich Maschinenbau mit einer solchen Aufgabe zu beauftragen, wäre ein Lösungsansatz, der jedoch eigene Probleme aufwirft. Denn die Aneignung des entsprechenden Fachwissens benötigt viel Zeit. Zudem fehlt oft das dazugehörige Grundlagenwissen. So könnte wertvolle Einsatzzeit des Maschinenbauers in seinem eigenen Fachbereich verloren gehen.

Zu beachten ist außerdem, dass die Einführung einer individuellen Datenerfassung meist deutlich aufwendiger ist, als deren anschließende Instandhaltung. Auch die Nutzung und das Management solcher Daten durch Personal ohne Fachwissen sind, abhängig vom gewählten System, meist unproblematisch. Nach der Implementierung wird ein großer Teil des Fachwissens also oft nicht mehr benötigt.

Das Problem einer individuellen Datenerfassung beläuft sich damit hauptsächlich, aber nicht nur, auf deren Implementierung. Insbesondere ist dies in Bereichen wie dem Maschinenbau der Fall, in denen die Informationstechnik (IT) als Schnittstelle dient und oft kein entsprechendes Fachwissen vorhanden ist. Problematisch ist dabei vor allem der Entwicklungsbereich zu sehen, da hier neben finanziellen Faktoren auch die Zeit eine große Rolle spielt, die benötigt wird, um ein solches System einzuführen.

1.2 Zielsetzung

In der Industrie sowie im Entwicklungsbereich ist es oft von Vorteil oder sogar notwendig, Sensoren zur Maschinenüberwachung in Verbindung mit Datenbanksystemen einzuführen. Die Anwendungsmöglichkeiten sind zahlreich. Sie können Testzwecken dienen aber auch die Anlagenüberwachung auf Dauer vereinfachen.

Ziel ist hier die Programmierung eines Webservices, der möglichst flexibel und übersichtlich aufgebaut ist. In Bezug auf die Problemstellung soll die Implementierung dieser Anwendung ohne großes IT-Fachwissen ermöglicht werden, sodass dies für jemanden aus anderen Fachbereichen, wie beispielsweise Maschinenbau, mit geringem Aufwand zu bewältigen ist.

Daneben soll der Nutzer in der Lage sein, auch ohne größere Programmierkenntnisse eine Datenbank zu erstellen, die anschließend durch den Webservice verwaltet wird. Falls bereits eine Datenbank besteht, soll die Anbindung an diese möglich sein.

Der Zugriff auf die erhobenen Daten soll über ein Netzwerk möglich sein. Dies erfordert einen entsprechenden Schutz vor unautorisierten Zugriffen. Neben der sicheren Verwaltung von Daten in einer Structured Query Language (SQL) Datenbank, wäre die Steuerung der Datenerfassung durch den Webservice und somit ebenfalls über das Netzwerk eine hilfreiche Funktion. Diese Steuerung soll, wie die Datenverwaltung selbst, nur für entsprechend autorisierte Personen oder Programme verfügbar sein.

Zudem soll die entsprechende Flexibilität und Komptabilität gegeben sein, um die API mit bestehenden Systemen verknüpfen zu können.

Schlussendlich soll sich eine Anwendung ergeben, die von einer Vielzahl von Systemen wie einem Browser, einer Handy-App oder durch ein eigenständiges Programm bedient werden kann. In kurzer Zeit sollen Daten erfasst und nach SQL-Schema durch eine API verwaltet, bereitgestellt und geschützt werden können. Außerdem soll die Möglichkeit der Steuerung der Datenerfassung durch die API bestehen. Trotz all dieser Funktionen, soll die Implementierung so einfach wie möglich ablaufen und ohne großes IT-Fachwissen zu bewältigen sein.

2 Grundlagentheorie

In diesem Kapitel sollen die theoretischen Grundlagen erklärt werden. Diese sind nötig, um die entworfenen Anwendungen, erläutert in den Kapiteln 3.2 und 3.3, nachvollziehen zu können. Beginnend mit dem allgemeinen Begriff des Webservices wird Schritt für Schritt das Grundlagenwissen bis hin zu expliziten Frameworks wie Flask oder Eve und Netzwerkkomponenten wie Nginx geschaffen. Dabei wird auch der Entscheidungsfindungsprozess erläutert, der zur Auswahl entsprechender Komponenten führte.

2.1 Webservice

Ein Webservice oder auch Webdienst ist eine über Netzwerke zugängliche Schnittstelle zu Anwendungsfunktionen (siehe Abbildung 1). Somit ist auch jede Anwendung, die mit Hilfe einer Kombination von Protokollen über ein Netzwerk angesprochen werden kann, ein Webservice (James Snell, 2002, S. 1).

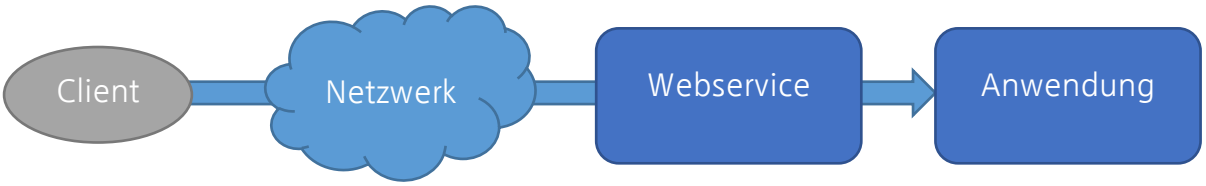

Abbildung 1: Webservice Schema

Wie in Abbildung 1 zu erkennen ist, dient ein solcher Webservice der Maschine-Maschine-Kommunikation. Er ermöglicht dem Client auf die Funktionen der Anwendung zuzugreifen und das unabhängig von der Sprache des Client. Der Client repräsentiert die Hard- beziehungsweise Software, die Anfragen sendet und Antworten empfängt. Prämisse ist dabei, dass der Webservice selbst die Sprache des Client unterstützt. Der Webservice stellt also eine Schnittstelle dar, die als Abstraktionsschicht wirkt. So können Anwendung und Client auf verschiedenen Sprachen basieren (zum Beispiel Browser in C++ und Anwendung in Python). Des Weiteren können sie auf verschiedenen Systemen laufen (Windows, Unix, Mac et cetera) und trotzdem problemlos miteinander kommunizieren beziehungsweise interagieren (James Snell, 2002, S. 2).

Webservices sind nachrichtenorientiert, dienen also der Übermittlung von Nachrichten. Sie sollten dazu eine Kombination von mehreren Standard-Internetprotokollen versenden und empfangen können. Ein typischer Einsatzzweck für Webservice ist es, Anwendungsfunktionen auf einem Server aufzurufen, nachdem dies von einem Client verlangt wurde. Der Client sendet also die Anfrage mit den entsprechenden Parametern und bekommt die Ergebnisse des Funktionsaufrufs zurückgesendet.

Abbildung 2 zeigt das vereinfachte Konzept eines Webservice. Wie in Abbildung 1 soll der Block „Anwendung“ stellvertretend für den gesamten Aufbau einer Anwendung stehen. Er enthält sowohl die Logik als auch den Code zum Ausführen bestimmter Funktionen.

Der Service-Listener nimmt die Anfragen entgegen, er muss also das Protokoll des Client beherrschen (meist HTTP oder HTTPS). Der Service-Proxy des Webservice hat die Aufgabe, die Anfragen so zu übersetzen und weiterzuleiten, dass die Anwendung die gewünschte Funktion ausführt (James Snell, 2002, S. 3)

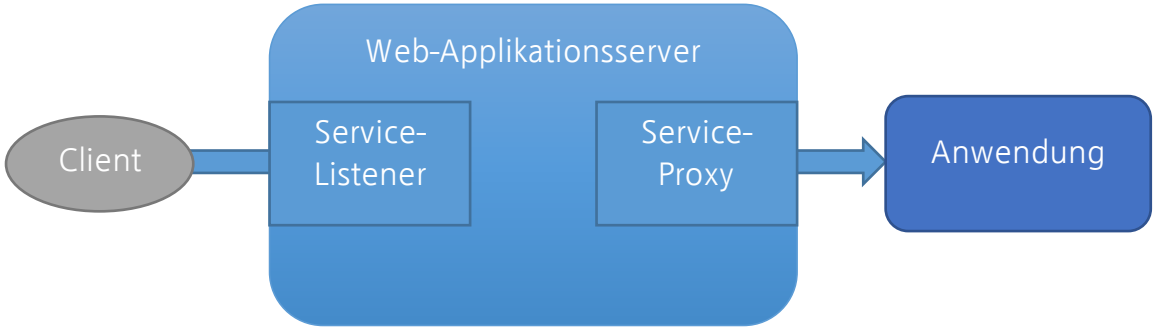

Abbildung 2: Service-Proxy und-Listener

Service-Proxy und –Listener können auf verschiedene Arten umgesetzt werden. Sie können aus eigenständigen Anwendungen bestehen, zum Beispiel mittels HTTP-Server als Service-Listener, aber auch gemeinsam innerhalb eines Applikationsservers laufen. Einige Webserver haben entsprechende Service bereits installiert.

2.1.1 Webservice-Architektur

Die Autoren des Buches Webservice-Programmierung mit SOAP (James Snell, 2002) beschreiben die Webservice-Architektur in Form von fünf Schichten (siehe Abbildung 3):

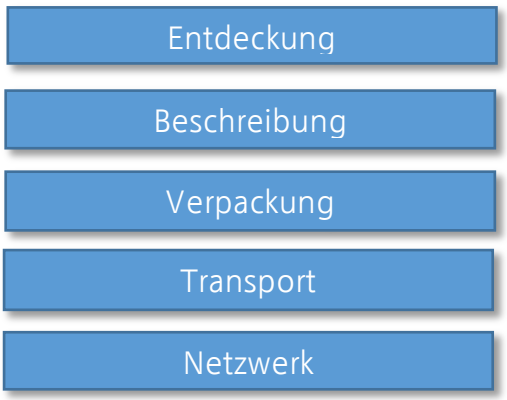

Abbildung 3: Webservice-Techniken

Die Entdeckungsschicht stellt sicher, dass der Client den Webservice finden kann. Dazu gibt es verschiedene Methoden wie beispielsweise Universal Description, Discovery and Integration (UDDI) (James Snell, 2002). UDDI ist ein Verzeichnisdienst der besonders dynamische Webdienste berücksichtigt.

Die Beschreibungsschicht gibt in der Web Service Description (WSD) Auskunft darüber, wie Nachrichten ausgetauscht werden. Die WSD definiert Nachrichtenformat, Datentypen, Transportprotokolle und Transportserialisierungsformate, die zwischen Client und Dienstanbieter genutzt werden. Diese Spezifikationen werden in der Web Service Description Language (WSDL) verfasst. Die Beschreibungsschicht stellt eine Abmachung dar, wie mit dem entsprechenden Service zu interagieren ist (W3C, 2017).

Die Verpackungsschicht „verpackt" die Anwendungsdaten, um diese im Netzwerk versenden zu können (auch Serialisierung genannt). Das Format muss dabei so gewählt werden, dass alle Beteiligten es verstehen können. Mögliche Verpackungsformate sind beispielsweise XML, JSON oder SOAP. Möglich wäre auch ein Format wie HTML, dies ist jedoch weniger geeignet, da es eher zur Darstellung als zur Deutung von Informationen dient (James Snell, 2002).

Die Transportschicht dient hauptsächlich dem Bewegen von Daten zwischen verschiedenen Punkten des Netzwerks. Verwendete Protokolle können das Transmission Control Protocol (TCP) aber auch HTTP oder SMTP et cetera sein. Die Wahl des Protokolls ist hauptsächlich von den Kommunikationsanforderungen des Webservice abhängig.

Die Netzwerkschicht stellt die grundlegenden Funktionen zur Kommunikation in einem Netzwerk zur Verfügung. Sie bietet die Möglichkeit zur Adressierung oder zum Routing.

2.2 HTTP

HTTP ist die Abkürzung für HyperText Transfer Protocol. Es ist ein zustandsloses Protokoll, eine Art Regelwerk zur Datenübermittlung in einem Computernetzwerk. Die Kommunikation baut auf dem Client-Server-Modell auf. Ein HTTP-Client, zum Beispiel ein User-Agent wie ein Browser, schickt dabei Anfragen (Request) an einen HTTP-Server, der diese entsprechend seiner Konfiguration verbarbeitet und beantwortet (Response) (Lauterschlag, 2017).

2.2.1 Request

Ein HTTP-Request besteht aus 3 Hauptteilen. Der erste Teil ist die Anforderung (siehe Code 1, Zeile 1). Darin wird zunächst die Request-Methode, dann der genaue Pfad zur gewünschten Ressource und schlussendlich die HTTP-Version angegeben.

Nach der Anforderung folgen die HTTP-Headerfelder. Für eine erfolgreiche Anfrage ist das Feld des Hosts (siehe Code 1, Zeile 2) zwingend notwendig. Weitere Felder wie Code 1 Zeile 3 dienen der Übermittlung von Parametern und Argumenten, welche die Kommunikation zwischen Server und Client erleichtern sollen.

```
1    GET /Pfad/zur/Ressource/ HTTP/1.1
2    Host: www.hostadresse.at
3    User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:39.0)
```

Code 1: HTTP-Request-Schema

Dem Request können zusätzlich Daten beigefügt werden, die der Client an den Server übermitteln möchte. Diese werden durch Leerzeichen von den Headerfeldern getrennt.

2.2.2 Response

Der HTTP-Server antwortet auf eine Anfrage mit einer HTTP-Response. Diese Antwort ist ähnlich zum Request aufgebaut. Die dreiteilige Status-Line in Zeile 1 (Code 2) besteht aus der verwendeten HTTP-Version, dem Statuscode und einer kurzen Erläuterung des Statuscodes. Der HTTP-Statuscode teilt dem Client mit, ob eine Anfrage erfolgreich war. Beziehungsweise gibt er Auskunft über mögliche Ursachen, falls Fehler aufgetreten sind (siehe Kapitel 2.2.4). In den folgenden Zeilen stehen Header-Informationen. Diese übermitteln, analog zu den Request-Headerfeldern, Parameter und Argumente.

```
1     HTTP/1.1 200 OK
2     content-type: gzip
3     date: Thu, 18 May 2017 15:09:14 GMT
```

Code 2: HTTP-Response-Schema

Die zu übermittelnden Daten, wie beispielsweise der HTML-Code einer Website, stehen nach den Header-Informationen und werden durch ein Leerzeichen von diesen getrennt.

2.2.3 Request-Methoden

HTTP unterstützt mehrere Methoden, die es dem Client ermöglichen Daten oder Informationen abzufragen beziehungsweise zu manipulieren. Tabelle 1 zeigt welche Request-Methoden es gibt und welche Funktion diese haben:

GET	Ressource abrufen.
POST	Daten an eine Ressource senden.
HEAD	Verursacht eine Abfrage wie GET, jedoch wird nur der Response-Header ohne die tatsächlichen Daten übermittelt.
PUT	Anlegen beziehungsweise Modifizieren einer Ressource.
DELETE	Löschen einer Ressource.
CONNECT	Veranlasst einen Proxy-Server eine Tunnelverbindung zur Verfügung zu stellen.
OPTIONS	Dient der Abfrage der vom Server unterstützten Request-Methoden.
TRACE	Liefert dem Client die Anfrage so zurück, wie der Server sie empfangen hat (dient hauptsächlich Diagnosezwecken).

Tabelle 1: Request-Methoden

2.2.4 HTTP-Status-Codes

Der dreistellige Status-Code gibt dem Client Auskunft darüber, was beim Nachrichtenaustausch passiert ist. Abgesehen von einem erfolgreichen Nachrichtenaustausch (Status-Code 2xx) können bei der Client-Server-Kommunikation auch etliche Dinge unplanmäßig verlaufen. Um auf ein besonderes Vorkommnis oder einen Fehler hinzuweisen gibt es deshalb Definitionen der entsprechenden Codes.

Die Status-Codes sind durch ihre erste Stelle in verschiedene Kategorien eingeteilt (siehe Tabelle 2). Der definierte Bereich kann wachsen, wenn neue Codes definiert werden. Die Kategorie gibt jedoch auch bei unbekannten Codes einen Anhaltspunkt über Art der Rückmeldung.

Code	Definierter Bereich	Kategorie
100-199	100-101	Information
200-299	200-206	Erfolgreich
300-399	300-305	Umleitung
400-499	400-415	Client-Error
500-599	500-505	Server-Error

Tabelle 2: HTTP-Status-Codes (Gourley, Totty, Sayer, Aggarwal, & Reddy, 2002)

Der Status-Code ist leicht für Maschinen zu interpretieren, um dem Nutzer die Deutung zu erleichtern wird jedem Code auch eine Nachricht beigefügt. Die wohl häufigsten Status-Codes sind in Tabelle 3 mitsamt Nachricht und genauer Bedeutung festgehalten (Gourley, Totty, Sayer, Aggarwal, & Reddy, 2002):

Code	Nachricht	Bedeutung
200	OK	Erfolg – die angefragten Daten sind in der Antwort des Servers.
401	Unauthorized	Es müssen Username und Passwort eingegeben werden.
404	Not Found	The Server kann die Ressource der angefragten URL nicht finden.

Tabelle 3: häufig auftretende HTTP-Status-Codes

2.3 SSL

Secure Socket Layer (SSL) wurde nach Version 3.0 unter neuem Namen als Transport Layer Security (TLS) weiterentwickelt, beginnend mit „TLS Version 1.0". SSL und TLS sind also Synonyme für das gleiche hybride Verschlüsselungsprotokoll, allerdings in verschiedenen Versionen (SSL Shopper, 2017). In dieser Arbeit wird der weitläufiger bekannte Name SSL verwendet.

Typisches Einsatzgebiet für SSL ist das Verschlüsseln sensibler Daten wie Bankzugangsdaten, Kreditkartennummern oder Login-Informationen. Diese Daten möchte ein Nutzer beispielsweise über seinen Browser zu einem Webserver senden. Da die Kommunikation im Internet meist durch die Übermittlung von Klartext erfolgt, kann jeder Computer der Daten weiterleitet diese lesen. So auch ein Angreifer der die Nachricht eines Nutzers auf deren Weg zum Webserver abfängt. Man spricht hier von einem Man-In-The-Middle-Angriff (SSL Shopper, 2017).

SSL erzeugt eine verschlüsselte Verbindung zwischen Server und Client und ist das Standardprotokoll, um Kommunikation im Internet zu schützen (CA Security, 2017). Das folgende Kapitel soll Einsatz und Funktionsweise des SSL-Protokolls zeigen.

2.3.1 Asymmetrische Verschlüsselung

Bei der asymmetrischen Verschlüsselung, oder auch Public-Key-Verschlüsselung, wird ein anderer Schlüssel zum Ver- als zum Entschlüsseln verwendet. Der Public-Key des Empfängers kann von jedem genutzt werden, um eine Nachricht zu verschlüsseln. Der dazugehörige Private-Key des Empfängers, der zur Entschlüsselung benötigt wird, ist geheim (siehe Abbildung 4). Typische Schlüssellängen sind 1024 bis 2048 Bits. Eine Verschlüsslung mittels 2048-Bit-Schlüssel gilt als sicher, da ein durchschnittlicher Computer heute mehrere Milliarden Jahre bräuchte, um diese ohne Private-Key zu entschlüsseln (DigiCert, 2017).

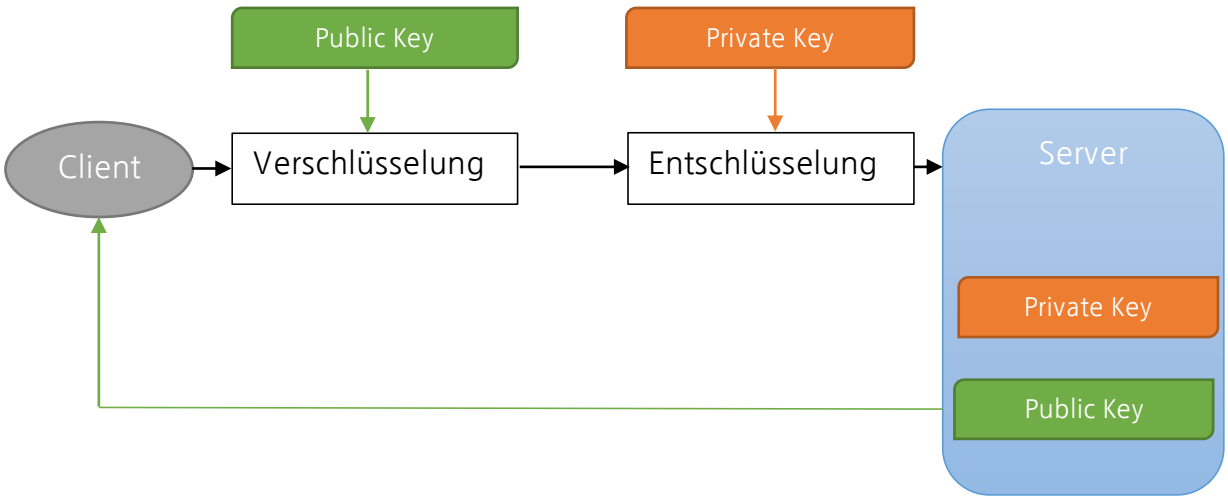

Abbildung 4: Asymmetrische Verschlüsselung

2.3.2 Symmetrische Verschlüsselung

Bei der symmetrischen Verschlüsselung gibt es nur einen einzigen Schlüssel mit dem sowohl die Ver- als auch die Entschlüsselung vollzogen wird. Sowohl der Sender als auch der Empfänger benötigen also den gleichen Schlüssel, der geheim bleiben muss (siehe Abbildung 5). Die Schlüssellänge symmetrischer Schlüssel liegt üblicherweise bei 128 oder 256 Bits. Generell gilt: Je länger der Schlüssel, desto sicherer ist die verschlüsselte Nachricht (SSL Shopper, 2017).

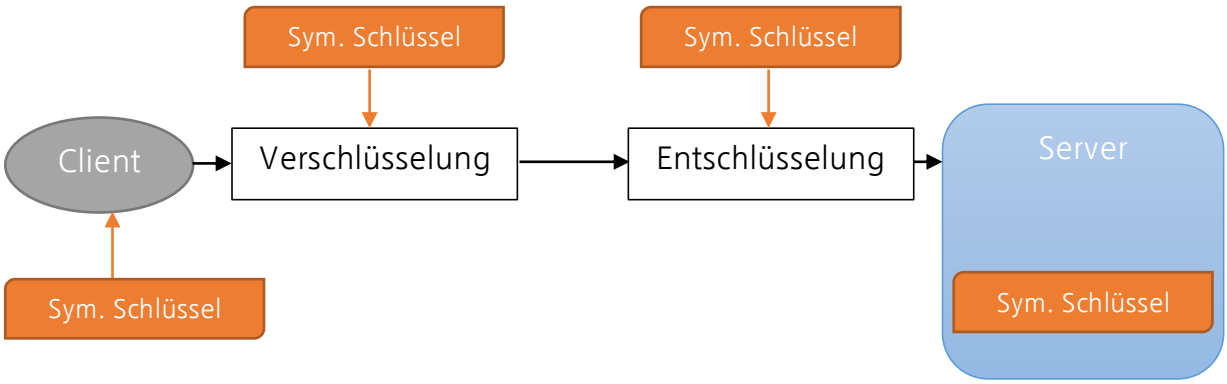

Abbildung 5: Symmetrische Verschlüsselung

2.3.3 SSL-Verschlüsselung

Beide Verfahren haben sowohl Vor- als auch Nachteile. Der längere Schlüssel der asymmetrischen Verschlüsselung macht es zwar unautorisierten Angreifern schwerer die Nachricht zu entschlüsseln, erhöht andererseits aber den Rechenaufwand.

Die symmetrische Verschlüsselung hingegen erlaubt die Ver- und Entschlüsselung mit geringem Rechenaufwand. Jedoch benötigen Sender und Empfänger den gleichen geheimen Schlüssel. Damit stellt die Distribution des symmetrischen Schlüssels ein großes Sicherheitsrisiko dar. Dieses Problem existiert bei asymmetrischer Verschlüsselung nicht, da nur der Public-Key versendet werden muss. Der Public-Key dient nur dem Ver-, nicht aber dem Entschlüsseln. Somit spielt es keine Rolle wer diesen Schlüssel erhält und es entsteht kein Sicherheitsrisiko durch dessen Übermittlung.

Bei der SSL-Verschlüsselung wird eine hybride Verschlüsselung genutzt, um die Vorteile von symmetrischen und asymmetrischen Systeme vereinen zu können. Dazu verfügt der Server über ein SSL-Zertifikat sowie über den Public- und den dazugehörigen Private-Key des asymmetrischen Systems. Der symmetrische Schlüssel wird im Laufe des SSL-Handshakes generiert.

Ablauf des SSL-Handshakes (Microsoft, 2017):

1. Der Client sendet seine SSL-Verbindungsinformationen an den Server
2. Der Server sendet seine SSL-Verbindungsinformationen und sein Zertifikat an den Client. Falls Clientauthentifizierung erforderlich ist fragt der Server außerdem das Clientzertifikat ab.
3. Der Client versucht den Server zu authentifizieren.
4. Der Client erstellt eine Vorstufe des geheimen Hauptschlüssels, verschlüsselt diese mithilfe des Public-Key des Servers und sendet sie an den Server. Falls Clientauthentifizierung stattfindet werden weitere vom Client signierte Daten und das Zertifikat des Client mitgesendet.
5. Im Falle von Clientauthentifizierung versucht der Server nun den Client zu authentifizieren und entschlüsselt anschließend die Vorstufe des Hauptschlüssels mithilfe seines Private-Key. Client und Server erzeugen nun den geheimen Hauptschlüssel.
6. Client und Server generieren den symmetrischen Sitzungsschlüssel mithilfe des geheimen Hauptschlüssels.
7. Der Client informiert den Server, dass weitere Kommunikation durch den Sitzungsschlüssel verschlüsselt wird. Es folgt die verschlüsselte Bestätigung des Client über den Abschluss des Handshakes.
8. Der Server informiert den Client, dass weitere Kommunikation durch den Sitzungsschlüssel verschlüsselt wird. Es folgt die verschlüsselte Bestätigung des Servers über den Abschluss des Handshakes.
9. Mit Ende des Handshakes beginnt die Sitzung. Verschlüsselte Kommunikation sowie Überprüfung der Datenintegrität erfolgt dann über den symmetrischen Sitzungsschlüssel.

Die oben erwähnten Server- und Clientauthentifizierungen stellen sicher, dass der jeweilige Kommunikationspartner auch tatsächlich ist, wer er vorgibt zu sein. Dies gelingt über das SSL-Zertifikat, das es in zwei Varianten gibt:

Das Self-Signed-Zertifikat wird meist für interne SSL-Verschlüsselung verwendet. Es ist kostenlos und leicht zu implementieren. Es bietet den gleichen Verschlüsselungsschutz wie ein von einer Zertifizierungsstelle signiertes Zertifikat. Jedoch gibt es keine vertrauenswürdige Auskunft über die Identität des Ausstellers.

Das Signed-Zertifikat wird von einer Zertifizierungsstelle, auch Certificate Authority (CA), ausgestellt. Dadurch wird neben dem Verschlüsselungsschutz noch eine Identifikation des Zertifikatinhabers gewährleistet (Dan, 2017).

2.4 SOAP

SOAP (ursprünglich das Akronym für Simple Object Access Protocol) ist ein standardisiertes Netzwerkprotokoll, mit dessen Hilfe Nachrichten zwischen Anwendungen ausgetauscht werden können. SOAP nutzt XML (siehe Abschnitt 2.4.1), um ein erweiterbares Messaging-Framework, also Rahmenbedingungen zum Umgang und Gestalten von Nachrichten, bereitzustellen. Dieses Framework wurde so entworfen, dass es sowohl vom Programmiermodell als auch von der Semantik anderer Implementierungen unabhängig funktionieren kann (W3C, 2017).

Da SOAP eine Anwendung der XML-Spezifikation ist, die sich in ihrer Definition und Arbeitsweise stark auf XML-Standards wie XML-Schema und -Namespaces stützt, soll XML im nächsten Abschnitt kurz erläutert werden.

2.4.1 XML

Extensible Markup Language (XML) ist eine Auszeichnungssprache zum Speichern und Transportieren von Daten. Ein Beispiel dafür, wie XML-Code aussehen kann, ist in Code 3 zu sehen. Die Informationen in XML-Code sind immer von Tags umschlossen (hier „<beispiel>...</beispiel>“, „<from>...</from>“ et cetera). Solche Tags sind bei XML nicht vorgegeben, der Verfasser des Codes kann also beliebige Tags verwenden. Auch die Struktur eines solchen XML-Dokuments kann der Verfasser selbst wählen. Ein XML-Code hat für sich genommen keine ausführbare Funktion. Stattdessen wird eine Software benötigt, um den Code zu senden, empfangen, speichern oder darstellen zu können. Diese Software muss die entsprechenden Tags dann jedoch deuten können. Dabei ist XML eher zum Datentransport entworfen, wohingegen HTML beispielsweise eine Auszeichnungssprache ist die eher der Darstellung dient (w3schools, 2017).

```
<beispiel>
  <from>Oliver</from>
  <heading>Beispielcode für XML</heading>
  <body>So könnte XML-Code aussehen</body>
  <freietags>Tags sind bei XML nicht vorgegeben</freietags>
</beispiel>
```

Code 3: XML-Beispiel

XML ist nicht an bestimmte Anwendungen, Betriebssysteme oder Programmiersprachen gebunden. Der XML-Nachrichtenaustausch, also der Austausch von Information zwischen Anwendung durch den Transport von XML-Dokumenten, ist somit sehr flexibel.

XML-Namespaces sollen Namenskonflikte von Elementen vermeiden. Sie dienen der eindeutigen Identifizierung von Elementen durch eine Uniform Resource Identifier (URI) Referenz (W3C, 2017). Da Elementnamen (Tags) bei XML frei wählbar sind, besteht die Gefahr von Konflikten, besonders wenn verschiedene XML-Dokumente gemischt werden. Code 4 und Code 5 geben ein Beispiel, wie ein solcher Konflikt aussehen kann. Würden diese zwei Codes in einem XML-Dokument zusammengefügt werden, entstünde ein Namenskonflikt, da das Element „table" sich hier auf zwei verschiedene Dinge bezieht.

```
<table>
  <sp>Spalte1</sp>
  <sp>Spalte2</sp>
</table>
```

Code 4: XML–Konfliktbeispiel, Tabelle

```
<table>
  <platte>Eiche</platte>
  <beine>Ahorn</beine>
  <preis>50,--</preis>
</table>
```

Code 5: XML–Konfliktbeispiel, Tisch

Ein solcher Namenskonflikt kann bei XML mithilfe von Präfixen (hier „a" und „b") gelöst werden. Das gemischte XML-Dokument, ohne Konflikt, kann dann wie in Code 6 gestaltet werden.

```
<a:table>
  <a:sp>Spalte1</a:sp>
  <a:sp>Spalte2</a:sp>
</a:table>


<b:table>
  <b:platte>Eiche</b:platte>
  <b:beine>Ahorn</b:beine>
  <b:preis>50,--</b:preis>
</b:table>
```

Code 6: XML-Präfix

Bei der Verwendung von Präfixen in XML wird zusätzlich ein Namespace (auch Namensraum) definiert. Eingeleitet wird der Namespace durch das XML-Namespace-Attribut (xmlns) im Start-Tag des Elements (siehe Code 7). Der zu verwendende Syntax dafür ist: xmlns:Präfix="URI"

```
<a:table xmlns:a="http://www.bsp.com/muss/nicht/existieren/Tabelle">
  <a:sp>Spalte1</a:sp>
  <a:sp>Spalte2</a:sp>
</a:table>


<b:table xmlns:b="http://www.bsp.com/muss/nicht/existieren/Tische">
  <b:platte>Eiche</b:platte>
  <b:beine>Ahorn</b:beine>
  <b:preis>50,--</b:preis>
</b:table>
```

Code 7: xmlns-Attribut

Der dabei verwendete URI dient dazu, dem Namespace einen eindeutigen Namen zu geben. Er wird nicht vom Parser genutzt, um Daten abzurufen oder zu verfolgen. Oft wird jedoch der Namespace als Pointer verwendet. Er verweist dann auf die Webseite, welche die Namespace-Informationen beinhaltet (W3C, 2017).

2.4.2 SOAP-Nachrichten

Abbildung 6 zeigt den strukturellen Aufbau einer SOAP-Nachricht, einem sogenannten SOAP-Envelope. Der Envelope kann einen Header enthalten, muss jedoch über einen Body verfügen. Im Header befinden sich Informationen zur Verarbeitung der Nachricht, wie zum Beispiel Routing, Authentifikation, Autorisierung oder Transaktionskontexte. Der Body enthält die Kerninformationen der Nachricht in XML-Syntax.

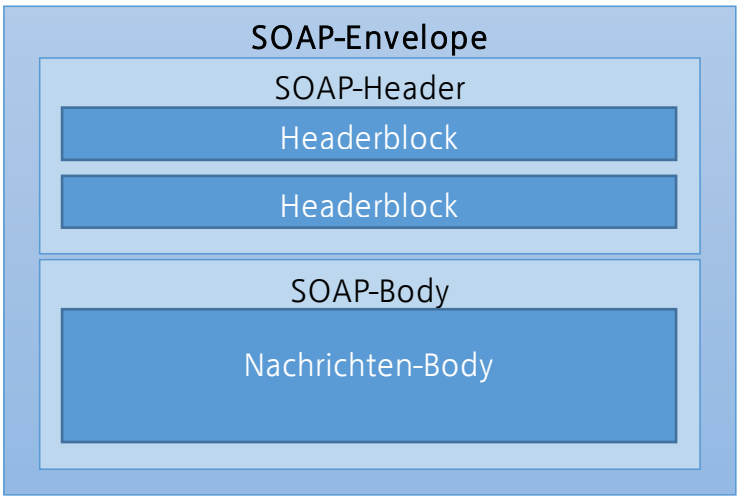

Abbildung 6: SOAP-Nachricht (James Snell, 2002)

Das Envelope-Element identifiziert das XML-Dokument als SOAP-Nachricht. Code 8 zeigt ein Beispiel für ein SOAP-Envelope-XML-Dokument. Zu beachten ist, dass der xmlns:soap-Namespace immer den Wert „http://www.w3.org/2003/05/soap-envelope/“ haben sollte. Ein anderer Wert an dieser Stelle kann eine Fehlermeldung aufwerfen. Eine Interpretation und Verarbeitung ist dann unter Umständen nicht möglich (w3schools, 2017).

```
<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
  <soap:Header>

  ...Headerblock...

  </soap:Header>

  <soap:Body>
  ...
      <soap:Fault>
      ...
      </soap:Fault>

  </soap:Body>
</soap:Envelope>
```

Code 8: SOAP-Envelope Beispiel (w3schools, 2017)

Jeder SOAP-Envelope darf maximal ein Header-Element enthalten. Der Header muss als erstes Kind vor dem Body stehen. Elemente innerhalb des Headers werden auch Headerblocks genannt. Diese können, ähnlich wie im Body, frei wählbare XML-Code-Elemente sein. Generell besteht ihre Aufgabe aber darin, Kontextinformationen zur Verarbeitung der SOAP-Nachricht zur Verfügung zu stellen.

Wie ein HTTP-Request mit SOAP aussehen kann und wie die dazugehörige Antwort aussehen könnte, zeigen Code 9 und Code 10 (näheres zu HTTP in Abschnitt 2.2). Der schwarzgedruckte Bereich des Codes gehört zu HTTP, der Rest entspricht der SOAP-Nachricht. In diesem Beispiel wird die Beispieladresse „www.bsp.org/NamesAndIDs“ angesprochen, um den User-Namen, der zu einer bestimmten UserID gehört, abzufragen. Dazu wird zunächst die entsprechende UserID (in diesem Beispiel 142) per HTTP-POST-Methode übermittelt:

```
POST /NamesAndIDs HTTP/1.1
Host: www.bsp.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn


<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

  <soap:Body xmlns:m="http://www.bsp.org/NamesAndIDs">
    <m:GetUserName>
      <m:UserID>142</m:UserID>
    </m:GetUserName>
  </soap:Body>

</soap:Envelope>
```

Code 9: SOAP HTTP-Anfrage

Wenn alles geklappt hat, sendet der Webservice den User-Namen (hier „Oliver“) per HTTP-Antwort zurück an den Client.

Dieses Beispiel soll nur den SOAP-Nachrichtenverkehr verdeutlichen. Die Anwendung (User-Name und –ID finden) ist eine Beispielanwendung und steht stellvertretend für alle möglichen Arten von Anwendungen, welche dem Datenaustausch dienen.

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

  <soap:Body xmlns:m="http://www.bsp.org/NamesAndIds">
    <m:GetUserNameResponse>
      <m:Name>Oliver</m:Name>
    </m:GetUserNameResponse>
  </soap:Body>

</soap:Envelope>
```

Code 10:SOAP HTTP-Antwort

2.5 REST[1]

Representational State Transfer (REST) beschreibt einen Architekturstil in der Informatik. Dieser wurde von Roy Thomas Fielding in seiner Dissertation „Architectural Styles and the Design of Network-based Software Architectures“ (Fielding, 2000) im Jahr 2000 definiert. Ziel war es dabei, einen Programmierstil zu entwerfen, der dem schnellen Wachstum des Internets gerecht werden kann.

2.5.1 Bedingungen

Bei der Gestaltung von REST geht Fielding von einem unbeschränkten System aus, für das die notwendigen Bedingungen entworfen werden. Diese Bedingungen und ihre Bedeutungen beschreiben die Grundlage der REST Architektur:

1. Client-Server

   Abbildung 7 zeigt das Schema der Client-Server Bedingung. Sie besagt, dass die Angelegenheiten von Client und Server getrennt sein sollen. User Interface (clientseitig) und Datenspeicherung (serverseitig) können sich damit unabhängig voneinander entwickeln.

Abbildung 7: Client-Server Bedingung

2. Stateless

   Weitere Bedingung ist die zustandslose Kommunikation zwischen Client und Server. Dabei soll jede Anfrage des Client an den Server alle benötigten Informationen enthalten. Der Zustand ist also nur über den Client gegeben und auf diesem gesichert. Dadurch erhöht sich zwar unter Umständen die Menge der zu sendenden Daten, dafür gestaltet sich die serverseitige Verarbeitung der Anfragen als übersichtlicher, leichter zu skalieren und zuverlässiger.

3. Cache

   Zur Steigerung der Netzwerkeffizienz wurde die Cache Bedingung eingeführt. Sie setzt voraus, dass Daten aus der Antwort des Servers als cacheable gekennzeichnet sind. Der Cache erlaubt sowohl dem Client, als auch dem Server die Speicherung von Daten für spätere Anfragen (falls diese als cacheable erkennbar sind). Damit können bestimmte Interaktionen teilweise oder sogar ganz

[1] Die Ausführungen in diesem Kapitel beruhen, wenn nicht anders vermerkt, weitgehend auf Roy Thomas Fieldings Dissertation (Fielding, 2000).

wegfallen. Somit können Effizienz und Leistung der Kommunikation erhöht werden. Dies geschieht jedoch eventuell auf Kosten der Zuverlässigkeit, denn Daten aus dem Cache können, zum Beispiel wenn sie veraltet sind, von den tatsächlichen Daten auf dem Server abweichen.

4. Uniform Interface

Einheitliche Schnittstellen (Uniform Interface) gehören zu den prägendsten Bedingungen des REST Architekturstils. Dabei werden Implementierungen von ihren eigentlichen Services getrennt und können sich so unabhängig vom Gesamtsystem entwickeln. Außerdem vereinfacht sich das Gesamtsystem und die Übersichtlichkeit der vorherrschenden Interaktionen erhöht sich. Dies geschieht jedoch zu gewissen Maßen auf Kosten der Effizienz, da Information nicht mehr in anwendungsspezifischer Form bereitgestellt werden können.
Zur Gestaltung dieser einheitlichen Schnittstellten gibt Fielding folgende vier Punkte an:

- Identification of Resources
  Grundsätzlich kann jede Art von Information eine Ressource darstellten. Falls ein Konzept jedoch Ziel einer Hypertext Reference ist, muss sie als Ressource behandelt werden. Jede Ressource soll mindestens eine spezielle Identifikationsadresse, den sogenannten Uniform Resource Identifier (URI), besitzen. Der URI bleibt gleich, auch wenn sich die Ressource ändert. Falls einer Ressource ein neuer URI zugewiesen wird, sollte der alte Identifier eine Fehlermeldung mit Verweis zum neuen zurückgeben (Aditya Mukerjee, 2017).

- Manipulation of Resources through Representations
  Falls ein Client Ressourcen auf dem Server ändern oder hinzufügen möchte, so geschieht dies indem er dem Server eine Darstellung (Representation) des geänderten Inhalts schickt. Der Server entscheidet dann ob die Ressource entsprechend geändert wird (Aditya Mukerjee, 2017).

- Self-descriptive Messages
  Self-descriptive Messages beinhalten alle Informationen, die nötig sind, um die Nachricht richtig zu interpretieren. Der Empfänger benötigt zur Vervollständigung der Nachricht also keine Informationen aus anderen Quellen (Aditya Mukerjee, 2017).

- Hypermedia As The Engine Of Application State (HATEOAS)
  HATEOAS bedeutet, dass der Client ausschließlich über Hypermedia mit dem Server interagiert. Bei der Kommunikation mittels Hypermedia beinhalten die Nachrichten des Servers an den Client immer Informationen darüber, welche weiteren Interaktionsmöglichkeiten dem Client zu Verfügung stehen (Aditya Mukerjee, 2017).

5. Layered System

Das Einführen von Schichten dient vor allem der Erhöhung der Skalierbarkeit. Dabei kann jede Systemkomponente nur die jeweils folgende Schicht erreichen, jedoch nicht direkt mit den Komponenten dahinter kommunizieren (siehe Abbildung 8). So können innerhalb einer Schicht neue Dienste getestet und durch die einheitlichen Schnittstellen ohne großen Aufwand an das Gesamtsystem angebunden werden. Beispiele für Komponenten wären Clients, Server, Gateways, Proxys.

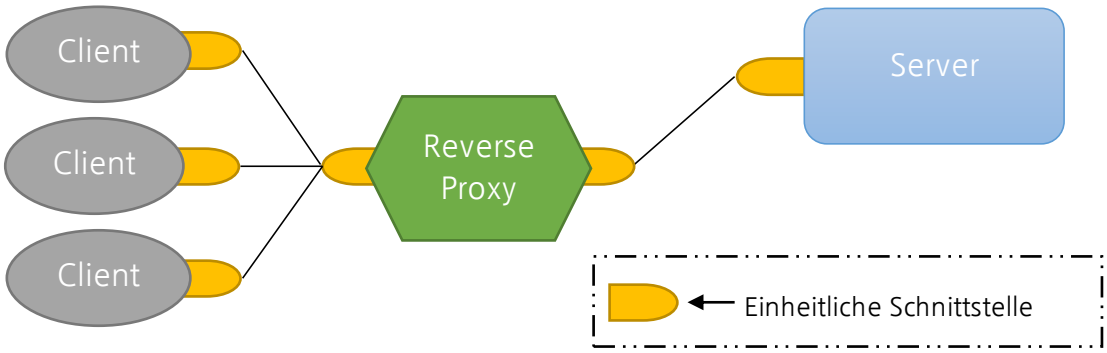

Abbildung 8: Layered System

6. Code-on-Demand

Die Code-on-Demand Bedingung erlaubt dem Client das Herunterladen von ausführbarem Code, wie in Applets oder Scripts. Zum einen kann so die Anzahl der benötigten vorinstallierten Funktionalitäten der Clients reduziert werden, zum anderen verbessert dies die Erweiterungsfähigkeiten des Systems. Allerdings verschlechtert Code-on-Demand die Übersicht.

Ob diese sechste Bedingung realisiert werden sollte oder nicht hängt von mehreren Voraussetzungen des Gesamtsystems ab. Oft ist es zum Beispiel aus sicherheitstechnischen Gründen nicht erwünscht, dass Java-Klassen heruntergeladen werden. Dies kann dann dazu führen, dass Code-on-Demand scheinbar nicht unterstützt wird.

2.5.2 Elemente

Neben den im vorigen Abschnitt erläuterten sechs architektonischen Bedingungen, führt Fielding drei architektonische Elemente ein. Diese gehören ebenso zur Basis des REST Architekturstils und lauten wie folgt:

1. Data Elements

   Art und Zustand eines Datenelements sind wichtige Bestandteile in der REST-Architektur. Bei der Bereitstellung von Information im hier vorherrschenden Hypermedia-Bereich gibt es drei grundsätzliche Optionen:

   - Die Daten am Speicherort rendern und anschließend in einem festgelegten Format an den Empfänger senden.
   - Die Daten mitsamt der Engine zum Rendern der Daten an den Empfänger senden.
   - Die Daten inklusive der zum Rendern benötigten Metadaten versenden, sodass der Empfänger selbst eine Engine zwecks Rendern wählen kann.

   Da jede der Methoden Vor- und Nachteile hat, bietet REST eine Art Mischung aus den drei Optionen. So liegt der Fokus auf Datentypen mit Metadaten, wobei der Umfang der Information durch das Erfassungsvermögen der standardisierten Schnittstelle begrenzt wird. Die Kommunikation zwischen den REST Komponenten findet dann, wie im vorherigen Kapitel beschrieben, durch das Versenden von Repräsentationen statt. Ob das Format der Repräsentation dabei von der Originalquelle stammt oder nicht, bleibt durch die Schnittstelle verborgen. Fielding definiert für REST folgende Datenelemente:

Datenelement	Beispiel
Resource	Jede Information, die zwischen Client und Server erreicht und versendet werden kann, wie zum Beispiel ein Bild auf einer Homepage.
Resource Identifier	URL
Representation	HTML Dokument, JPEG Bild
Representation Metadata	Internet Media Type (Image, Text, Video et cetera), Änderungszeit
Resource Metadata	Source link
Control Data	Cache-Control

Tabelle 4: Data Elements

2. Connectors

Die verschiedenen Connector-Typen (siehe Tabelle 5) dienen der Kommunikationsverwaltung zwischen den einzelnen Komponenten. Sie erlauben den Zugriff auf Ressourcen und das Versenden von Repräsentationen. Durch den Einsatz einer solchen Schnittstelle können Komponenten leicht geändert, getauscht oder hinzugefügt werden, da der Connector die Netzwerkkommunikation übernimmt.

Connector	Beispiel
Client	libwww, libwww-perl
Server	Apache API, Nginx
Cache	Browser Cache
Resolver	bind (DNS lookup library)
Tunnel	SOCKS, SSL

Tabelle 5: Connector

3. Components

Die vier REST Components (siehe Tabelle 6) bestehen aus interaktionsfähiger Software und sind nach deren jeweiligen Rollen kategorisiert.

Component	Beispiel
Origin Server	Webserver wie Apache httpd, Microsoft IIS
Gateway	Reverse Proxy wie Nginx, Squid, CGI
Proxy	Proxy-Server wie CERN Proxy, Gauntlet
User Agent	Webbrowser wie Lynx, Google Chrome

Tabelle 6: Components

2.5.3 Architektonische Sicht

Es soll nun gezeigt werden, wie die in den vorigen Kapiteln erläuterten Bausteine der REST-Architektur miteinander verknüpft sind und interagieren. Dazu beleuchtet Fielding den Aufbau aus drei Sichten:

- Process View
  Die Prozesssicht zeigt vor Allem die Beziehungen zwischen den Komponenten. Dies geschieht durch Visualisierung des Datenflusses (siehe Abbildung 9).

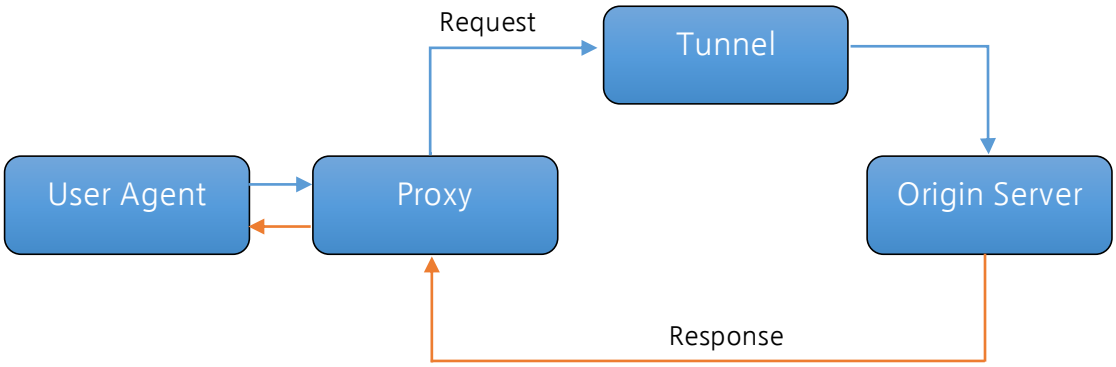

Abbildung 9: Process View

Der User Agent sendet einen Request, der alle vorgegebenen Komponenten auf seinem Weg zum Server durchläuft. Da die von REST vorgeschriebene Trennung von Client und Server die Komponenten-Implementierung vereinfacht und die Komplexität der Connector-Semantik reduziert, können ohne großen Aufwand an jeder Stelle Komponenten eingefügt werden. Diese erfüllen unterschiedlichste Zwecke, wie das Erhöhen der Sicherheit (zum Beispiel durch Firewalls), Traffic Balancing oder die Reduktion der Latenz. Da der Informationsfluss zwischen Client und Server komplett dynamisch ist, kann außerdem der Weg des Request ein anderer als jener der Response-Nachricht sein. Weiter benötigen die Komponenten keine Informationen über die Topologie des Gesamtnetzwerks, um Nachrichten auszutauschen (Puglisi, 2015).

- Connector View
  Die Connector-Sicht zeigt die Vorgänge bei der Kommunikation zwischen den Komponenten. Dabei liegt das Augenmerk auf den REST-Schnittstellen. So beschränkt sich REST nicht auf ein bestimmtes Protokoll, sondern legt in den Schnittstellen lediglich den Spielraum für Interaktionen und die Implementierungsvorgaben fest.
  Diese Aufgabe muss also nicht durch die Komponenten selbst erledigt werden. REST ermöglicht damit eine nahtlose Verbindung auch zwischen Unterschiedlichen Protokollen.

- **Data View**
  Die Datensicht beschäftigt sich mit dem Zustand einer Anwendung während des Datenaustauschs durch die verschiedenen Komponenten. Da REST auf verteilte Informationssysteme ausgerichtet ist, werden Anwendungen als zusammenhängende Struktur von Informationen und Kontrollalternativen gesehen. Diese führen die vom Benutzer geforderten Aufgaben aus. Eine der häufigsten Aufgaben im Internet ist der Abruf von Repräsentation verschiedener Ressourcen (zum Beispiel durch einen HTTP GET Request, siehe Abbildung 10).

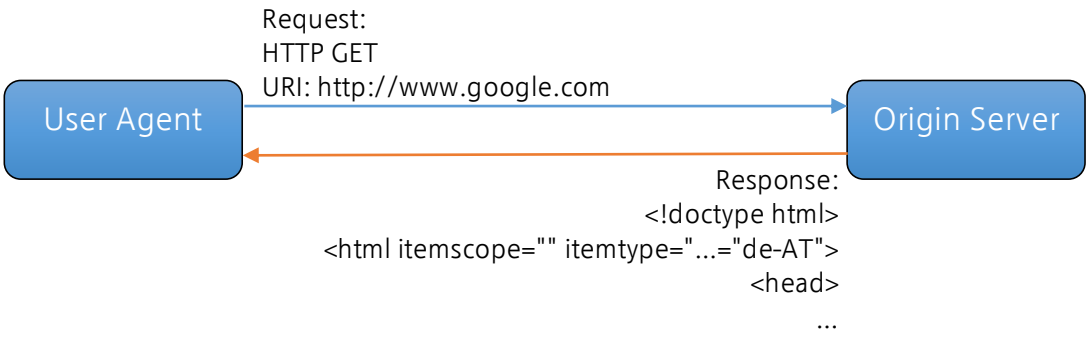

Abbildung 10: Beispiel GET Request

Um die Serverskalierbarkeit zu erhöhen, ist das Ziel der Gestaltung, dass der Server den Zustand des Client nicht über die aktuelle Anfrage hinaus kennen muss. Somit ist der Anwendungszustand gegeben durch:

- Laufende Anfrage
- Topologie der verbundenen Komponenten
- Aktive Anfrage an den Komponenten
- Datenfluss durch die Repräsentationen in der entsprechenden Antwort
- Verarbeitung der Repräsentationen auf Seite des User Agent

Die Anwendung befindet sich also im stationären Zustand solange keine Anfrage verarbeitet wird. Wie schnell dabei eine Anwendung vom stetigen Zustand über die Bearbeitung einer Anfrage wieder in den stetigen Zustand zurückwechseln kann, determiniert die vom Nutzer empfundene Performance der Anwendung.

2.5.4 JSON

REST schreibt die Verwendung von JavaScript Object Notation (JSON) zwar nicht vor, jedoch wird JSON in vielen Fällen XML vorgezogen (siehe 2.6.3). Ein grobes Verständnis von JSON wird benötigt, um die Entscheidungsfindung im Laufe dieser Arbeit nachvollziehen zu können. In diesem Abschnitt wird deshalb ein Überblick geschaffen.

JSON ist ein Datenaustauschformat. Es ist für den Menschen leicht zu lesen und zu schreiben, auch die Syntaxanalyse (Parsing) durch Maschinen ist schnell und effizient. JSON ist ein Textformat mit vielen Konventionen, ist aber von der Programmiersprache unabhängig. Es baut auf zwei Strukturen auf (Smith, 2015):

- **Name-Wert-Paare** (in den meisten Sprachen realisiert als Objekte, Sätze, Strukturen, Dictionaries et cetera)
- **Geordnete Listen mit Werten** (in den meisten Sprachen realisiert als Arrays, Vektoren, Sequenzen oder Listen)

Ein **Objekt** besteht aus einer ungeordneten Menge von Name-Wert-Paaren, die nach dem Schema in Abbildung 11 gebildet werden. Ein solches Objekt beginnt immer mit einer geöffneten geschwungenen Klammer und endet mit einer geschlossenen geschwungenen Klammer. Das Objekt kann leer sein (oberer Pfad), aus einem Name-Wert-Paar bestehen (mittlerer Pfad) oder mehrere Paare beinhalten, die dann von einem Komma getrennt werden (json, 2017).

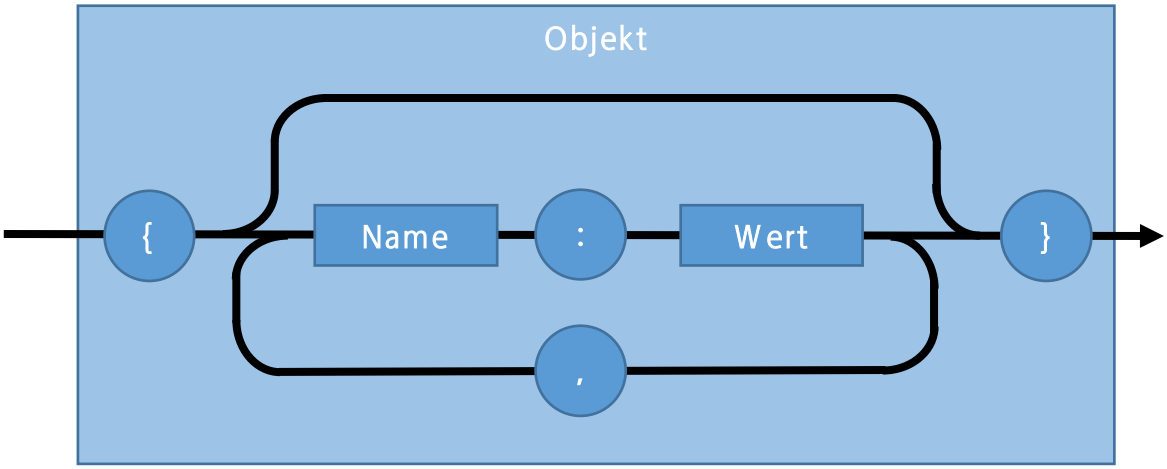

Abbildung 11: JSON-Objekt-Schema

Abbildung 12 zeigt den Aufbau einer geordneten Liste beziehungsweise eines **Array**. Analog zum Objekt kann das Array leer sein, aus einem, oder aus mehreren Werten bestehen. Ein solches Array wird von eckigen Klammern umschlossen.

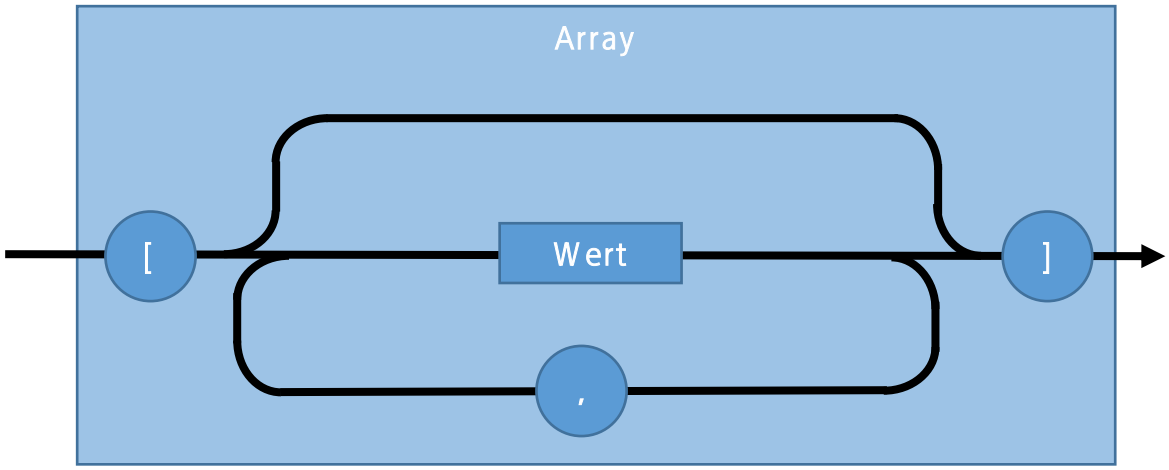

Abbildung 12: JSON-Array-Schema

Neben Werten wie „true“, „false“, „null“ kann ein Wert ein String, eine Zahl sowie ein Objekt oder Array selbst sein. So können Arrays und Objekte ineinander und in sich selbst geschachtelt sein. Abbildung 13 stellt die Detailansicht der Wert-Blöcke aus Abbildung 11 und Abbildung 12 dar:

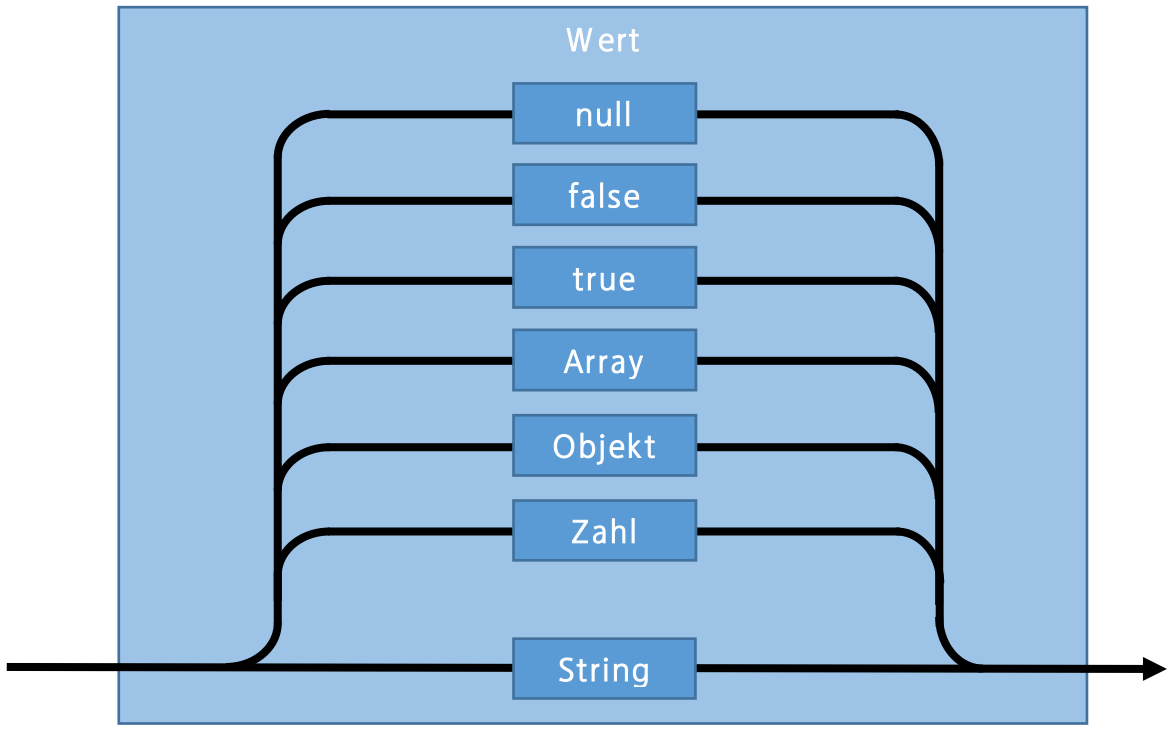

Abbildung 13: Details JSON Wert

Tabelle 7 zeigt die Möglichkeiten des String-Blocks, beziehungsweise unter welchen Bedingungen Eintragungen an dieser Stelle gemacht werden können. Generell kann eine beliebige Anzahl von Unicodezeichen, eingeschlossen in Anführungszeichen, an dieser Stelle stehen. Es lassen sich außerdem mit dem Backslash als Maskierungszeichen die in der Tabelle aufgelisteten Zeichen einfügen (Smith, 2015):

Unicode	Funktionszeichen	Maskierungszeichen	Interpretation
u0022	"	\"	Anführungszeichen
u005c	\	\\	Backslash
u002F	/	\/	Slash
u0008	b	\b	Backspace
u000C	f	\f	Seitenumbruch
u000A	n	\n	Zeilenumbruch
u000D	r	\r	Carriage Return
u0009	t	\t	Tab
uXXXX	uXXXX	\uXXXX	Unicodezeichen

Tabelle 7: JSON Maskierungszeichen

2.6 REST versus SOAP

SOAP und REST sind die zwei wesentlichen Standards im Webservice-Bereich (Zwattendorfer, 2013). Nachdem in den vorherigen zwei Kapiteln, 2.4 und 2.5, die Funktionsweisen von SOAP und REST erläutert wurden, soll nun untersucht werden, welches der beiden Systeme für den Anwendungsfall dieser Arbeit besser geeignet ist.

Folgend werden die Vor- und Nachteile von SOAP und REST erläutert. Dabei wird Bezug auf den Anwendungsfall dieser Arbeit genommen, da die Effektivität von SOAP und REST stark vom jeweiligen Verwendungszweck abhängt.

2.6.1 Vorteile von SOAP

SOAP ist Sprach-, Plattform- und Transportunabhängig. Im Gegensatz zu REST ist SOAP nicht an HTTP gebunden. Da HTTP jedoch zu den weitverbreitetsten Dateiübertragungsprotokollen zählt, ist es in diesem Anwendungsfall aus Gründen der Kompatibilität, einfachen Verständlichkeit und guten Dokumentation sinnvoll HTTP zu verwenden. Die Unabhängigkeit von HTTP stellt hier also keinen echten Vorteil von SOAP zu REST dar.

SOAP bietet gute Erweiterbarkeit durch Protokolle, Programme und Funktionen. Dazu gehören Standards wie WS-Security, ein Kommunikationsprotokoll zur Berücksichtigung von Sicherheitsaspekten, WS-Addressing zum Austausch von Adressinformationen, WS-Coordination zur Koordination von Aktionen verteilter Anwendungen, WS-ReliableMessaging zur Stabilisierung des Nachrichtenversands und viele mehr.

2.6.2 Vorteile von REST

Einer der größten Vorteile von REST gegenüber SOAP ist die Möglichkeit JSON (siehe 2.5.4) nutzen zu können. SOAP hingegen ist an XML gebunden. Viele der Vorteile von REST entstehen durch die Verwendung von JSON.

Mitunter durch den Einsatz von JSON bietet REST eine schnellere Syntaxanalyse (auch Parsing genannt) bei leichterem Umgang mit dem Webservice (Stringfellow, 2017).

REST bietet bessere Performance als SOAP, da es weniger Bandbreite benötigt und schnellere Übertragung ermöglicht. Dies ist zum einen durch die geringere Datengröße von JSON gegenüber XML möglich, zum anderen durch den Einsatz von geschickt gewähltem Caching (Stringfellow, 2017).

REST ist in den meisten Fällen leichter in bestehende Website-Strukturen zu implementieren (Stringfellow, 2017).

2.6.3 Fazit

Es gibt Situationen, in denen es sinnvoller ist SOAP zu nutzen, hier jedoch überwiegen die Vorteile die REST für den vorliegenden Anwendungsfall bieten kann. Zu dieser Entscheidung führten vor allem die höhere Flexibilität und die Einsatzmöglichkeit von JSON gegenüber XML. Denn nicht nur die Performance, auch die Verständlichkeit und Übersicht erscheint bei JSON höher als bei XML (siehe XML-Code-Beispiel in Code 11 und Code 12). JSON benötigt keine End-Tags und unterstützt außerdem die Verwendung von Arrays.

```
<tische>
  <tisch>
    <platte>Eiche</platte>
    <beine>Ahorn</beine>
    <preis>50,--</preis>
  </tisch>
  <tisch>
    <platte>Marmor</platte>
    <beine>Esche</beine>
    <preis>250,--</preis>
  </tisch>
  <tisch>
    <platte>Plastik</platte>
    <beine>Plastik</beine>
    <preis>5,--</preis>
  </tisch>
</table>
```

Code 11: XML-Code-Beispiel

```
{"table":[
    { "platte":"Eiche", "beine":"Ahorn" , "preis":"50,--" },
    { "platte":"Marmor", "beine":"Esche" , "preis":"250,--" },
    { "platte":"Plastik", "beine":"Plastik" , "preis":"5,--" }
]}
```

Code 12: JSON-Code-Beispiel

2.7 Reverse Proxy Server

Ein Reverse Proxy Server ist eine Kommunikationsschnittstelle im Netzwerk, die Anfragen entgegennimmt und entsprechend weiterleitet. Ein solcher Reverse Proxy Server sitzt dabei meist im privaten Netzwerk und leitet die Anfragen der Clients an den gewünschten Backend Server weiter (siehe Abbildung 14). So bietet ein Reverse Proxy Server eine zusätzliche Ebene der Abstraktion und Kontrolle (NGINX, 2017).

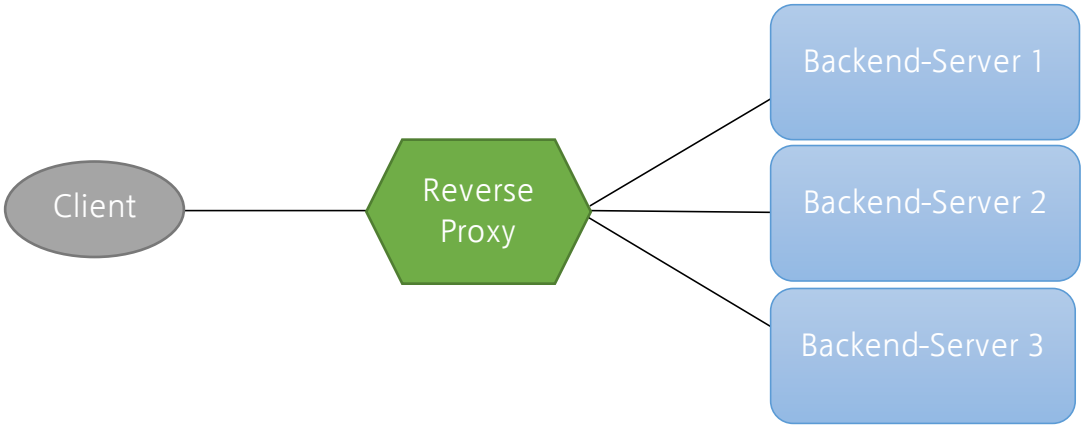

Abbildung 14: Reverse Proxy Server

Das Load Balancing ist ein typischer Verwendungszweck eines solchen Reverse Proxy Servers. Dabei agiert der Server als Lastverteiler und sorgt für eine Performance-orientierte Weiterleitung der Client-Requests zu den Backend-Servern. Wenn also im Beispiel von Abbildung 14 der Backend-Server 1 überlastet wäre, würden weitere Anfragen durch den Load-Balancer an die Backend-Server 1 und 2 geleitet werden (NGINX, 2017).

Durch die Fähigkeit, von Reverse Proxy Servern eingehende und ausgehende Daten zu komprimieren und oft genutzte Daten zu cachen, können sie den Datenverkehr zwischen Client und Server beschleunigen. Des Weiteren übernehmen sie oft die Aufgabe der SSL Verschlüsselung (dazu mehr in Kapitel 2.3) und entlasten damit die Backend-Server. So kann die Ansprechzeit der Server verkürzt und der Reverse Proxy Server als Web Accelerator genutzt werden (NGINX, 2017).

Da der Reverse Proxy Server die Client-Anfragen entgegennimmt, erhöht er außerdem die Sicherheit und Anonymität, indem die Backend-Server nicht direkt zu erreichen sind. Zudem können sie alle über eine URL, nämlich die des Reverse Proxy Servers, angesprochen werden.

2.8 Apache

Der Apache HTTP Server wurde 1995 von Robert McCool geschaffen und wird seit 1999 durch die Apache Software Foundation weiterentwickelt. Seit 1996 gehört der Apache Server zu den beliebtesten Servern im Internet. Durch die weite Verbreitung ist er außerdem gut dokumentiert (Ellingwood, 2017).

Um Funktionen zu ergänzen, kann der Apache Web Server über Module erweitert werden. Dazu existieren über 60 offizielle Module von Apache und eine Vielzahl von inoffiziellen Modulen.

2.8.1 Verbindungsarchitektur

Apache bietet verschiedene Multi-Processing Module (MPM), die einen unterschiedlichen Umgang mit Client-Anfragen bieten. So kann je nach Bedarf eine andere Architektur der Verbindungsgestaltung erreicht werden:

- mpm_prefork: Dieses Modul arbeitet mit mehreren Prozessen mit jeweils einzelnen Threads. Jeder dieser Prozesse kann nur eine Anfrage gleichzeitig behandeln. Solange es mehr Prozesse als Anfragen gibt, ist die Bearbeitungsdauer der Anfragen kurz (Ellingwood, 2017). Bei einer größeren Anzahl von Anfragen kann es jedoch zu Problemen und sogar zur Ablehnung von Verbindungen kommen.

- mpm_worker: Dieses Modul arbeitet mit Prozessen die mit mehreren Threads arbeiten können. Jeder der Threads kann eine Verbindung handhaben. Da Threads effizienter sind als Prozesse, ist die Skalierbarkeit des mpm_worker Moduls besser als die des mpm_prefork Moduls. Da jedoch ein einzelner Steuerprozess die Verbindungsprozesse und Verbindungen verwaltet, kann es auch hier bei vielen Anfragen zu Problemen kommen.

- mpm_event: Dieses Modul ähnelt in einigen Bereichen dem mpm_worker Modul, wurde jedoch für sogenannte keep-alive-Verbindungen (persistente Verbindungen) optimiert. Im Gegensatz zum Worker-Modul unterscheidet das Event-Modul langlebige Verbindungen von anderen. Persistente Verbindungen werden an speziell dafür bereitgestellte Threads weitergeleitet. Dieses Modul ist das Standardmodul, vorausgesetzt das Betriebssystem unterstützt es (Ellingwood, 2017).

2.9 Nginx

Nginx ist ein HTTP (siehe Kapitel 2.2) und Reverse Proxy Server, der ursprünglich von Igor Sysoev entwickelt wurde. Veröffentlicht wurde Nginx im Jahr 2004. Seine weite Verbreitung (Netcraft, 2017) verdankt es dem Ruf, hohe Performance, Stabilität, ein reichhaltiges Feature-Set, einfache Konfiguration und ressourcenschonenden Betrieb zu bieten (Soni, 2016, S. 1).

Nginx kann durch Module erweitert werden, verfügt jedoch von vorneherein über eine Vielzahl von Funktionen wie:

- Reverse Proxy Server für HTTP, HTTPS, SMTP, POP3 und IMAP Protokolle
- Load Balancing und HTTP Cache
- Frontend Proxy für Web Server
- FastCGI, uwsgi und SCGI Unterstützung zur Verarbeitung dynamischer Inhalte wie in PHP oder Python Skripten

2.9.1 Verbindungsarchitektur

Nginx ist generell so entworfen, dass es asynchrone, non-blocking, eventgesteuerte Verbindungen verwendet. Dazu generiert es Worker-Prozesse, von denen jeder mehrere tausend Verbindungen regeln kann. Ermöglicht wird dies durch eine schnell wiederkehrende Überprüfung auf Prozess-Events. Die Entkopplung der Arbeit, die durch das Verwalten von Verbindungen entsteht, ermöglicht es dem Worker sich nur um Verbindungen kümmern zu müssen wenn es auch tatsächlich notwendig ist.

Wenn eine Verbindung von einem Worker verwaltet wird, wird diese in einer Event-Schleife platziert. Innerhalb dieser Schleife werden Events asynchron behandelt (das Senden und Empfangen von Nachrichten bei asynchroner Kommunikation kann zeitlich versetzt sein. Es wird also nicht zwingend auf Antwort des Kommunikationspartners gewartet). So kann dem Blockieren von Verbindungen vorgebeugt werden. Nach dem Schließen der Verbindung wird diese aus der Schleife entfernt (Ellingwood, 2017).

Durch diese Art der Verbindungsbearbeitung ermöglicht Nginx eine gute Skalierbarkeit bei geringem Ressourcenaufwand. Der Server kann mit einem einzelnen Thread arbeiten, wobei nicht für jede neue Verbindung ein neuer Prozess generiert werden muss. So bleibt die CPU- und RAM-Auslastung auch bei starkem Betrieb relativ konsistent (Ellingwood, 2017).

2.10 Apache versus Nginx

Da Nginx und Apache die zwei dominierenden Server im Internet sind (Netcraft, 2017), sollen deren Vor- und Nachteile für den Anwendungsfall dieser Arbeit nun abgewogen werden. Nachdem in den vorigen Kapiteln ein grober Einblick in die Funktionsweise und Strukturen von Apache und Nginx geschaffen wurde, kann nun genauer auf deren Unterschiede eingegangen werden.

2.10.1 Behandlung statischen Inhalts

Bei statischen Inhalten oder Webseiten ändert sich der Inhalt sehr selten (zum Beispiel ein Artikel auf Wikipedia). Sie stellen oft ein HTML-Dokument auf einem Webserver dar.

Bei reiner Behandlung von statischen Inhalten ist Nginx deutlich schneller als Apache. In einem Test von Aurimas Mikalauskas (Mikalauskas, 2017) konnte Nginx beinahe die doppelte Anzahl an Anfragen pro Sekunde verarbeiten als der Apache Server:

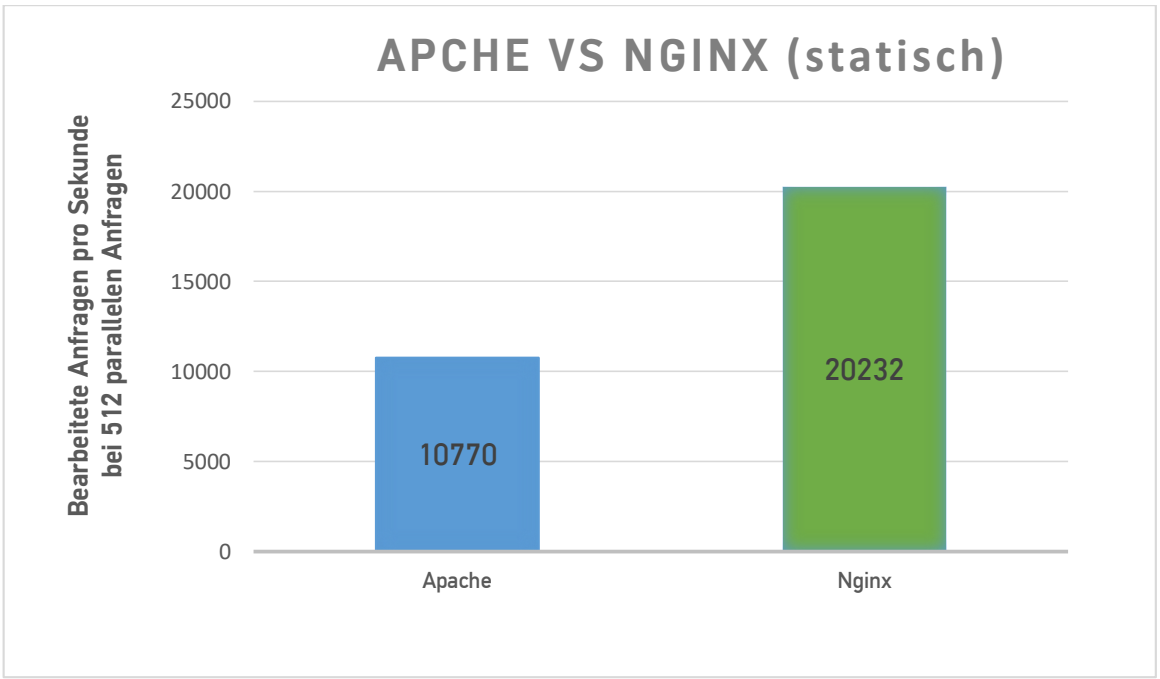

Abbildung 15: Apache versus Nginx statischer Inhalt

In diesem Test wurde ein 150kb großes JPEG Bild bei steigender Anzahl paralleler Abfragen abgerufen. Neben der deutlich besseren Performance von Nginx, arbeitete es zusätzlich effizienter als Apache (nämlich mit 11,8% statt 15,5% Speicherauslastung des Testsystems). Zu beachten ist dabei, dass es sich um rein statischen Inhalt handelt. Hieraus können keine direkten Rückschlüsse auf dynamische Inhalte geschlossen werden (Mikalauskas, 2017).

2.10.2 Behandlung dynamischen Inhalts

Bei dynamischen Inhalten oder Webseiten ändert sich der Inhalt sehr häufig (Wetter, Aktienkurse et cetera) oder ist direkt von einer Nutzeranfrage abhängig, wie zum Beispiel bei einer Suchmaschine.

Der Apache Server bietet die Möglichkeit, dynamische Inhalte zu verwalten. Dazu wird eine Erweiterung in die Worker-Instanzen eingefügt, die das Verarbeiten der entsprechenden Sprache erlaubt. Durch die interne Behandlung dynamischen Inhalts ist die Konfiguration oft einfacher als bei der Verarbeitung des dynamischen Inhalts durch eine externe Anwendung (auch dies ist bei Apache mittels entsprechender Module möglich).

Bei Nginx ist keine interne Verarbeitung dynamischer Inhalte möglich. Nginx gibt diese an eine andere Anwendung ab und muss dann auf deren Rückmeldung mit dem entsprechenden Ergebnis warten. Deshalb unterstützt Nginx Sprachen, wie FastCGI SCGI, uwsgi et cetera, die eine schnelle Kommunikation mit entsprechenden Schnittstellen von externen Anwendungen ermöglichen.

Abbildung 16 zeigt die Ergebnisse des Test von Aurimas Mikalauskas (Mikalauskas, 2017) zur Bearbeitungseffizienz dynamischer Anfragen an Nginx und Apache Server. Hier ergeben sich keine Performance- oder Effizienzunterschiede. Die dynamischen Anfragen benötigen deutlich länger als die statischen. Die Server können jedoch in der Zeit in der sie auf die Bearbeitung der dynamischen Anfrage (zum Beispiel durch PHP_FPM) warten, weiter statische Anfragen bearbeiten, ohne extra Prozesse starten zu müssen.

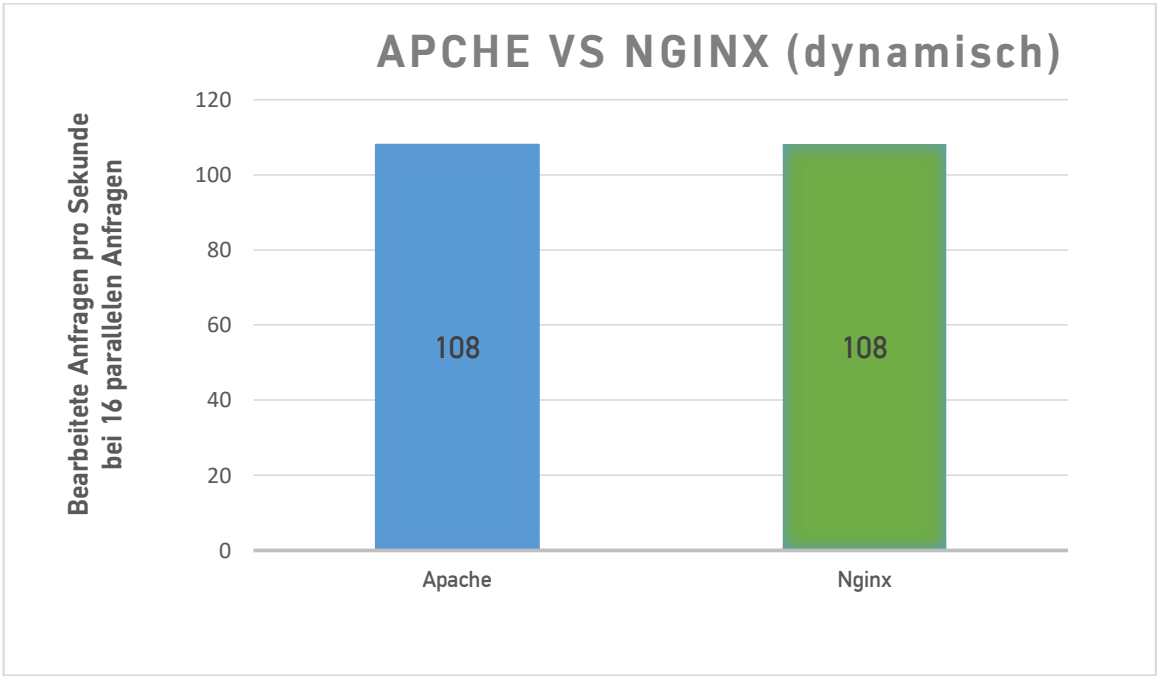

Abbildung 16: Apache versus Nginx dynamischer Inhalt

2.10.3 URI- und dateibasierte Interpretation

Da Apache als Webserver aufgebaut wurde, ist die Standardinterpretation von Ressourcenanfragen dateibasiert. Die Ressource wird also mittels Dateipfad gesucht und identifiziert (Ellingwood, 2017).

Apache bietet Alternativen, falls eine Anfrage nicht das zugrundeliegende Dateisystem bedienen kann. So kann eine Ressource auch an anderen Orten abgefragt und durch den Server bereitgestellt werden.

Apache besitzt also die Möglichkeit, sowohl mit einem Dateisystem als auch mit Webspace zu arbeiten, tendiert aber in seinem Design und Aufbau zur Verwendung eines Dateisystems (Ellingwood, 2017).

Nginx wurde mit dem Hintergrund gestaltet, nicht nur als Webserver, sondern auch ein Proxy zu arbeiten. Aufgrund dieser zwei Rollen, die Nginx erfüllen soll, arbeitet es hauptsächlich mit URI-basierter Interpretation und übersetzt dies in eine dateibasierte wenn nötig. Diese Eigenschaft macht es Nginx leichter als Proxy-Server eingesetzt zu werden (Ellingwood, 2017).

2.10.4 Module

Wie in den Erläuterungen zu Apache und Nginx schon erwähnt, lassen sich beide Systeme durch Module erweitern. Wie diese Module für die beiden Server zur Verfügung gestellt werden, unterscheidet sich jedoch.

Das Modulsystem von Apache erlaubt es Module bei laufendem Server zu aktivieren und zu deaktivieren. So können Funktionen in laufendem Betrieb nach Bedarf hinzugefügt oder entfernt werden.

Da der Apache Server schon seit vielen Jahren im Internet genutzt wird, existiert eine große Anzahl an Modulen. Diese ergänzen nicht nur Funktionen, sondern ändern auch bestehende Funktionen ab.

Beispiele für Funktionen die solche Module bereitstellen sind unter anderem:

- Client-Authentifizierung
- Caching
- Proxying
- Verschlüsselung
- Logging (Protokollierung von Vorgängen, meist zur Fehlersuche)
- Datenkompression

Im Unterschied zu Apache können die Module bei Nginx nicht dynamisch geladen werden. Es ist also nicht möglich sie im laufenden Betrieb hinzuzufügen oder zu entfernen. Stattdessen müssen die Module in die Nginx-Kernsoftware kompiliert werden. Für einige Nutzer wird Nginx dadurch unflexibler, besonders wenn sie Module benötigen die nicht in einem Distributionspaket enthalten sind. Denn dann muss der Server von der Basis aus selbst gestaltet werden. Es existieren jedoch viele Distributionen welche die Standardmodule bereits enthalten (Ellingwood, 2017).

Nginx-Module erlauben ähnliche Funktionen wie Apache-Module. Beispiele dafür sind:

- Proxying
- Datenkompression
- Logging
- Geolocation
- Authentifizierung
- Verschlüsselung

2.10.5 Fazit

Bei genauerer Betrachtung von Nginx und Apache erkennt man schnell, dass beides sehr fähige System sind, die ihre Berechtigung haben. Die Vor- und Nachteile müssen also für den entsprechenden Anwendungsfall abgewogen werden.

In Bezug auf Performance bietet Nginx deutliche Vorteile. Vor allem die ressourcenschonende Effizienz ist für den Anwendungsfall dieser Arbeit wichtig, um auch bei Systemen mit geringer Leistung, wie dem Einplatinencomputern Raspberry Pi, gute Werte in Bereich Leistung und Geschwindigkeit zu erzielen.

Auch das URI-basierte System von Nginx entspricht viel eher dem vorliegenden Anwendungsfall als das dateibasierte System von Apache. Zwar ist Apache in der Lage URI-basierend zu arbeiten (siehe Abschnitt 2.10.3), es benötigt jedoch eine entsprechende Konfiguration die das System verkomplizieren kann.

In Bezug auf Flexibilität durch den Einsatz von Modulen bietet Apache generell das bessere System. Da Nginx in der Standarddistribution jedoch alle Module für den Anwendungsfall dieser Arbeit enthält, fällt dieser Vorteil von Apache hier nicht ins Gewicht.

Für den Zweck dieser Arbeit wurde sich aus den oben genannten Gründen für Nginx entschieden.

Es sei noch angemerkt, dass eine Kombination von Apache und Nginx auch möglich ist. Meist wird Nginx als Frontend Reverse Proxy Server genutzt (so auch im gegenständlichen Anwendungsfall), der die entsprechenden Anfragen dann an den Apache Server weiterleitet. So kann eine gute Performance mit guter Flexibilität beziehungsweise leichter Einbindung in bestehende Systeme vereint werden.

2.11 WSGI

Das Web Server Gateway Interface (WSGI) ist ein Python Standard, Python Enhancement Proposal (PEP) 3333, und wurde 2003 von Philip J. Eby eingeführt. Es ist eine Spezifikation zur Beschreibung der Kommunikation zwischen Web-Server und Web-Applikationen/Framework. Es soll eine simple und universelle Schnittstelle zwischen Anwendung und Web-Server darstellen.

Die WSGI Schnittstelle hat eine Server-/Gateway-Seite und eine Application-/Framework-Seite. Außerdem können Middleware Komponenten implementiert werden.

2.11.1 Application/Framework-Seite

Das Application-Objekt ist eine aufrufbare Funktion, Methode, Klasse oder Instanz mit einer __call__-Methode. Dieses Objekt muss in der Lage sein mehrmals aufgerufen zu werden, um wiederholte Serveranfragen verarbeiten zu können. PEP 3333 gibt dafür das Beispiel einer solchen Funktion und Klasse (Code 13) (Eby, 2017).

```
HELLO_WORLD = b"Hello world!\n"

def simple_app(environ, start_response):
    """Simplest possible application object"""
    status = '200 OK'
    response_headers = [('Content-type', 'text/plain')]
    start_response(status, response_headers)
    return [HELLO_WORLD]

class AppClass:
    """Produce the same output, but using a class

    (Note: 'AppClass' is the "application" here, so calling it
    returns an instance of 'AppClass', which is then the iterable
    return value of the "application callable" as required by
    the spec.

    If we wanted to use *instances* of 'AppClass' as application
    objects instead, we would have to implement a '__call__'
    method, which would be invoked to execute the application,
    and we would need to create an instance for use by the
    server or gateway.
    """

    def __init__(self, environ, start_response):
        self.environ = environ
        self.start = start_response

    def __iter__(self):
        status = '200 OK'
        response_headers = [('Content-type', 'text/plain')]
        self.start(status, response_headers)
        yield HELLO_WORLD
```

Code 13: WSGI Application-Objekt Beispiel (Funktion und Klasse) (Eby, 2017)

2.11.2 Server/Gateway-Seite

Die Server/Gateway-Seite soll so gestaltet werden, dass jedes Mal, wenn ein HTTP Client eine Anfrage an die Anwendung schicken möchte, die Server/Gateway-Seite das Application-Objekt abruft (Eby, 2017).

2.11.3 Middleware

Als Middleware werden Komponenten bezeichnet, die beide Seiten, also Server- und Application-Seite, bedienen. Typische Funktionen einer solchen Middleware sind (Eby, 2017):

- Routing eines Request an verschiedene Anwendungsobjekte basierend auf der Ziel-URL
- Mehrere Frameworks oder Anwendungen im gleichen Prozess nebeneinander laufen lassen
- Load Balancing und Remote Processing durch die Weiterleitung von Anfragen und Antworten in einem Netzwerk
- Verschiedene Arten des Postprocessing

Sowohl die Server/Gateway- als auch die Application/Framework-Seite sollten erkennen können, ob Middleware-Komponenten vorhanden sind.

2.12 uWSGI versus Gunicorn

Zwei häufig verwendete, und für Flask sowie Eve empfohlene Server sind Gunicorn und uWSGI (Grinberg, 2014, S. 221). Deshalb sollen diese zwei Systeme in diesem Kapitel nach kurzer Vorstellung miteinander verglichen werden.

Zunächst soll eine Begrifflichkeit geklärt werden, die leicht zu Verwechslungen führt: uWSGI (großgeschrieben) ist der Name des Servers, wohingegen uwsgi (kleingeschrieben) das Protokoll bezeichnet.

Der **uWSGI** Server ist so gestaltet, dass er als Schnittstelle zwischen Nginx, dem HTTP-Server und der Anwendung fungiert. Außerdem unterstützt er das Übermittlungsprotokoll uwsgi.

Gunicorn ist ein WSGI- und HTTP-Server der dem gleichen Zweck dient wie der uWSGI-Server. Gunicorn hat keine uwsgi Unterstützung und nutzt stattdessen meist das WSGI-Protokoll zur Kommunikation mit Python-Anwendungen.

Kurt Griffiths hat in einem Performance-Test von Gunicorn und uWSGI dabei folgende Ergebnisse erzielt (Griffiths, 2017):

Abbildung 17 zeigt den Durchsatz der beiden Systeme. Dabei ist deutlich zu erkennen, dass uWSGI im vorliegenden Testfall wesentlich mehr Anfragen pro Sekunde verarbeiten kann als Gunicorn. Gunicorn scheint hier bei ca. 500 Verbindungen an die Belastungsgrenze zu kommen.

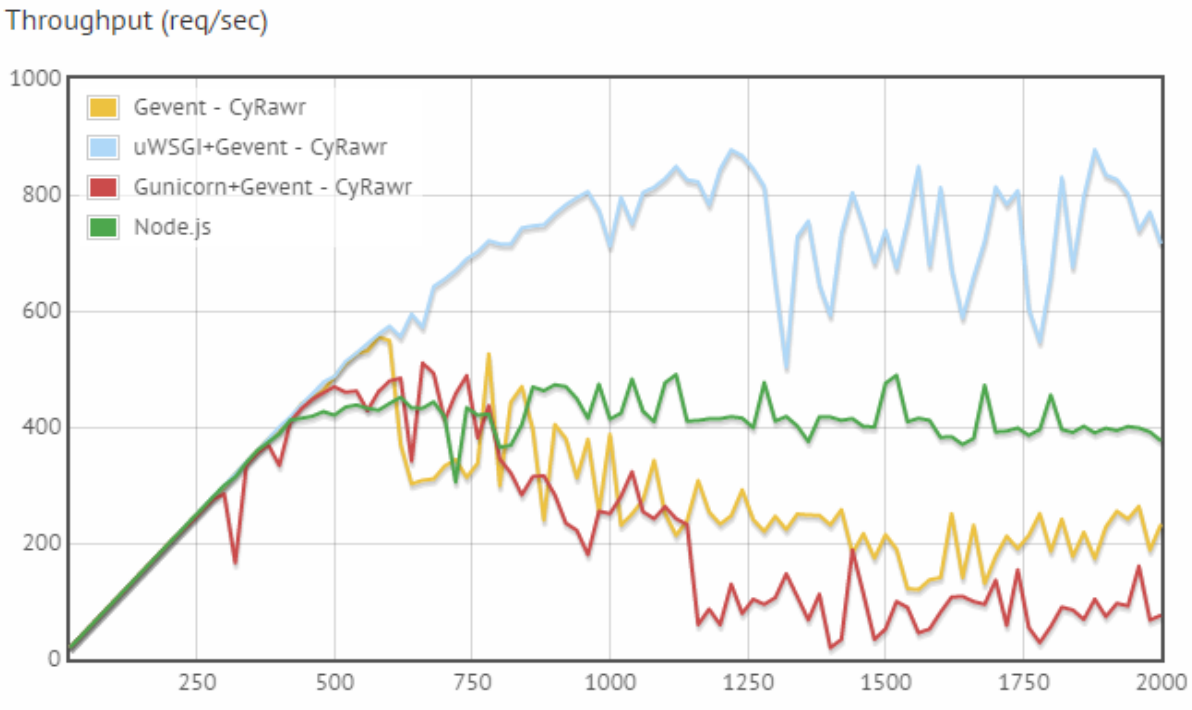

Abbildung 17: Durchsatz Gunicorn versus uWSGI (Griffiths, 2017)

uWSGI bietet unter den gegebenen Testparametern auch bessere Antwortzeiten als Gunicorn (siehe Abbildung 18). Bei 2000 Verbindungen ist uWSGI nahezu doppelt so schnell wie Gunicorn.

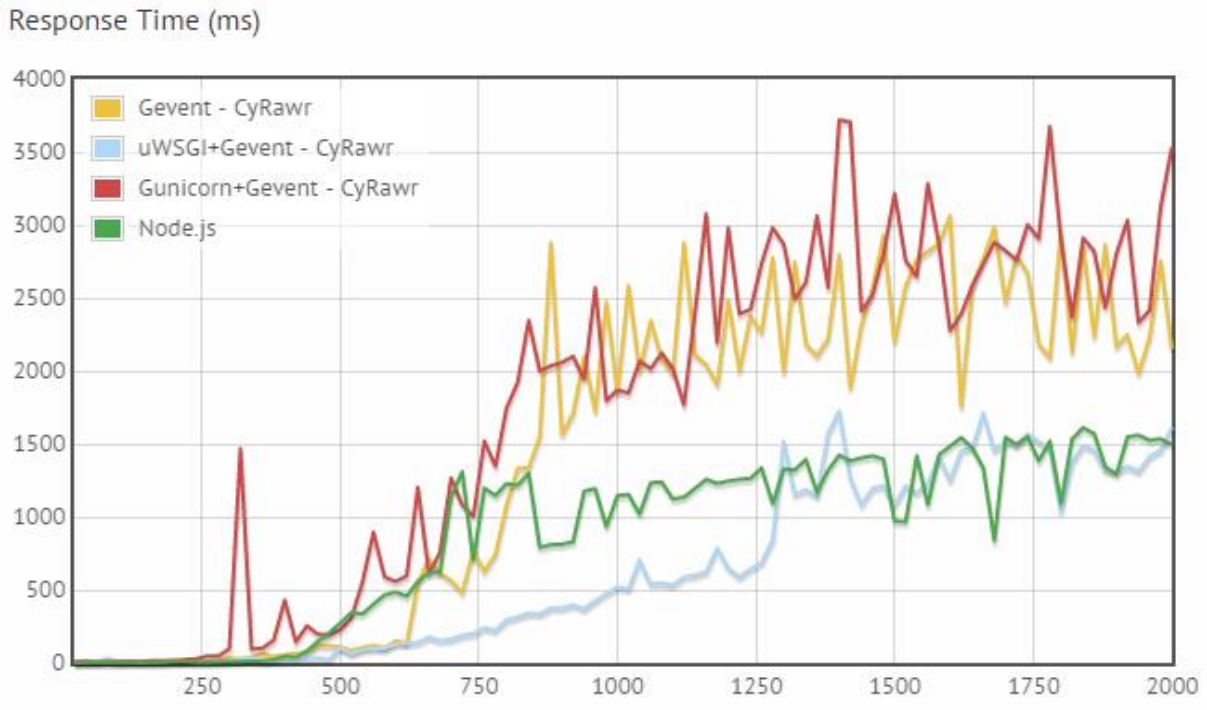

Abbildung 18: Antwortzeit Gunicorn versus uWSGI (Griffiths, 2017)

Auch im Test auf Fehleranfälligkeit liefert uWSGI bessere Ergebnisse als Gunicorn (siehe Abbildung 19). So scheint uWSGI den Betrieb nahezu unabhängig von der Verbindungsanzahl fehlerfrei zu gewährleisten, wohingegen Gunicorn bei ungefähr 1000 Verbindungen beginnt, deutlich mehr Fehler zu produzieren.

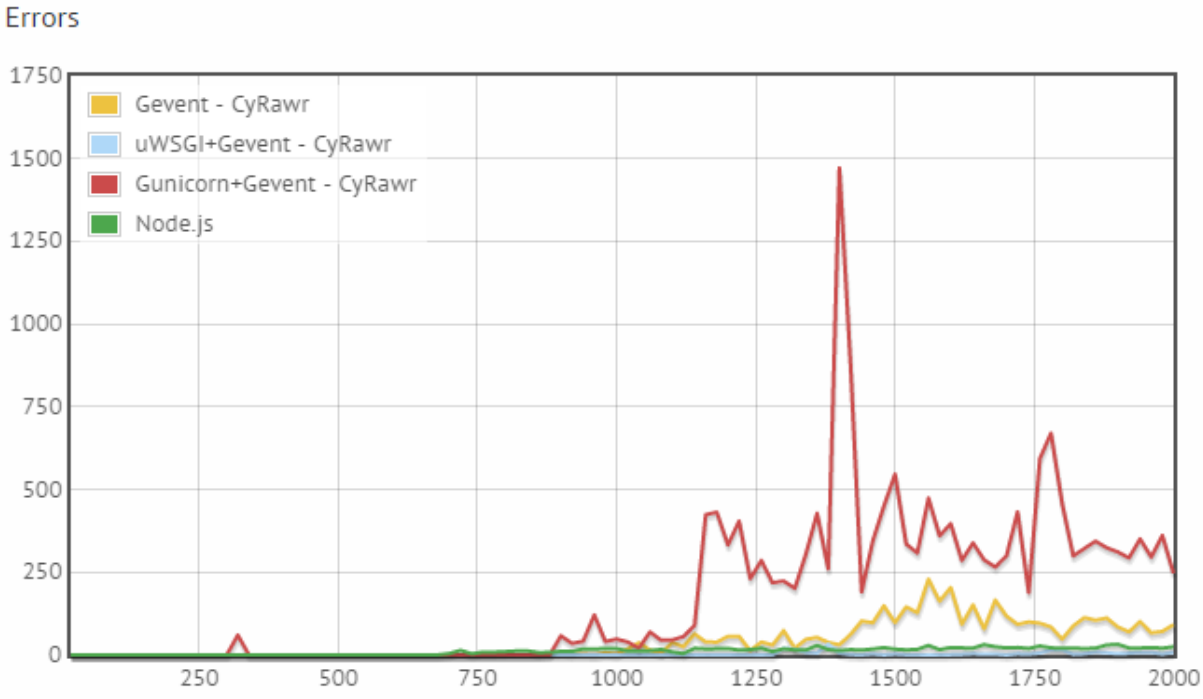

Abbildung 19: Fehler Gunicorn versus uWSGI (Griffiths, 2017)

Laut diesem Test bietet uWSGI also deutlich bessere Performance-Eigenschaften als sein Vergleichspartner Gunicorn.

In Bereichen der Konfiguration oder Schlankheit beider Systeme konnten keine Vorteile gefunden werden, die für den Anwendungsfall dieser Arbeit ins Gewicht fallen würden. Somit fiel die Entscheidung uWSGI Gunicorn vorzuziehen.

2.13 Frameworks

Ein Framework ist kein eigenständiges Programm, sondern bietet dem Entwickler eine Art Programmiergerüst, mit dessen Hilfe er eine Anwendung erstellen kann. So existieren Frameworks für die unterschiedlichsten Anwendungsbereiche. Im Kontext dieser Arbeit liegt das Augenmerk besonders auf Web Application Frameworks (WAF). Ein WAF liefert das Grundgerüst zur Erstellung von dynamischen Webseiten, Webservices oder Webanwendungen.

Solche Frameworks können vordefinierte Klassen liefern, die zum Beispiel bei der Implementierung von Nutzerauthentifikation, Login-Vorgängen, Email-Versand oder Endpunktrealisierung helfen können. Zur Gestaltung der API des Anwendungsfalls dieser Arbeit wurde somit ein Framework gesucht, das die Grundgerüste der benötigten Funktionen möglichst übersichtlich zur Verfügung stellt. Das folgende Kapitel soll einige Frameworks, die dafür in Betracht kamen kurz erläutern und schlussendlich in Bezug auf den vorliegenden Anwendungsfall bewerten.

2.13.1 Flask

Flask präsentiert sich als Micro Webdevelopment Framework für Python. Micro bezieht sich im Fall von Flask auf den Basiskern des Frameworks. Dieser beschränkt sich nämlich mit knapp 10.000 Zeilen Code (Dwyer, 2017) auf die wesentlichen Funktionen einer API. Alle zusätzlich gewünschten Funktionen müssen über Erweiterungen hinzugefügt werden.

So gibt es unter Flask ohne Erweiterungen keine Unterstützung zur Datenbankanbindung, Nutzerauthentifikation oder ähnlichen komplexen Aufgaben. Daraus ergibt sich der Vorteil einer möglichst schlanken API, die bei entsprechender Ergänzung der Erweiterungen die meisten gewünschten Aufgaben bewältigen kann. Dabei spart sie sich Funktionen, die für den entsprechenden Anwendungsfall nicht benötigt werden (Grinberg, 2014).

2.13.2 Ruby on Rails

Ruby on Rails (kurz auch Rails) ist ein Open Source Web Application Framework, das in der Programmiersprache Ruby geschrieben ist. Öffentlich vorgestellt wurde es erstmals im Jahr 2004. Einer der Grundgedanken dabei war es Rails so zu gestalten, dass es alles beinhaltet was es benötigt, um das Programmieren von Webanwendung einfacher zu machen (Ruby on Rails, 2017).

Rails basiert auf der Philosophie zweier Hauptprinzipien:

- Don’t Repeat Yourself (DRY)
  DRY ist ein Prinzip in der Software-Entwicklung, bei dem versucht wird Redundanzen zu vermeiden oder zumindest auf ein Minimum zu beschränken. Informationen sollen eine einzige, eindeutige, maßgebende Darstellung innerhalb eines Systems haben. Durch DRY ist der Code weniger fehleranfällig und gleichzeitig leichter zu erweitern und zu warten (Ruby on Rails, 2017).

- Convention Over Configuration
  Rails ist so gestaltet, dass es Standardkonfigurationen besitzt, die sich nach Konventionen im Web-Anwendungsbereich richten. So soll die Komplexität von Konfigurationen reduziert werden (Ruby on Rails, 2017).

2.13.3 Django

Django ist ein Open Source Python Web Framework, das bereits im Jahr 2005 veröffentlicht wurde (Hourieh, 2009). Es zielt vor allem auf die schnelle Entwicklung von Web-Applikationen ab. Es findet seinen Grundstein im Zeitungs- und Nachrichtenbereich, in dem oft möglichst schnell Web-Anwendungen veröffentlicht werden mussten (Django, 2017).

Django basiert, ähnlich wie Ruby on Rails, auf dem DRY-Prinzip (siehe Abschnitt 2.13.2).

Um in entsprechender Geschwindigkeit das Programmieren einer fähigen API zu ermöglichen, bietet Django ein sehr umfangreiches Set an bereits implementierten Funktionen beziehungsweise Anwendungen. Dazu zählen unter anderem:

- Stand-Alone Webserver
- Serialisierung und Validierung zur Datenbankanbindung
- Caching Framework mit mehreren Caching-Methoden
- Template-System

Diese vielen und teilweise sehr komplexen Funktionen ermöglichen die schnelle Entwicklung von APIs, da nur selten weitere Funktionen implementiert oder selbst geschrieben werden müssen. Sie führen aber auch zu einem großen Kernprogramm von ungefähr 240.000 Zeilen Code (Dwyer, 2017). Wenn dem Anwender dann nicht klar ist, welche Funktion welche Aufgabe übernimmt, kann dies unübersichtlich und verwirrend wirken.

2.13.4 Spring

Spring ist ein Open Source Framework für die Java Plattform, speziell J2EE. Es dient also der Entwicklung von Java Anwendungen. Die erste öffentliche Version, Spring 1.0, erschien im Juni 2003.

Spring ist als Modulares System aufgebaut, bei dem die Entkopplung der Applikationskomponenten eine zentrale Rolle spielt.

Allgemeine Vorteile die für Spring sprechen sind zum Beispiel (Spring, 2017):

- Java Methoden können Datenbanktransaktionen ausführen, ohne auf eine Transaktions-API zurückgreifen zu müssen
- Java Methoden können als HTTP-Endpunkte dienen, ohne, dass sie die Servlet API benötigen
- Java Methoden können ohne die JMS API als Message Handler eingesetzt werden
- Java Methoden können als Management Operation arbeiten, ohne die JMX API nutzen zu müssen

Spring basiert auf den Grundsätzen der aspektorientierten Programmierung (AOP). AOP ist eine Ergänzung der objektorientierten Programmierung (OOP). Im Gegensatz zur OOP, bei der die Schlüsseleinheit der Modularität die Klasse ist, ist dies bei AOP der Aspekt. Es können also technische Aspekte isoliert und vom eigentlichen Programmcode getrennt werden. Dazu nutzt Spring eine seiner Hauptkomponenten, das AOP Framework. Dies muss jedoch nicht zwingend genutzt werden (Spring, 2017).

2.13.5 Eve

Eve ist ein Open Source Python REST API Framework zur Programmierung hoch anpassungsfähiger, vollfunktionaler RESTful Web Services. Es basiert auf Flask in Verbindung mit Cerberus und nutzt MongoDB als natives Datenbanksystem. Hauptziel von Eve ist es, Nutzern gespeicherte Daten über eine RESTful Web API zur Verfügung zu stellen (Iarocci, 2017).

In Abschnitt 2.13.1 wurden bereits die Grundsätze des Flask Frameworks erklärt. Eve besitzt alle Fähigkeiten die auch Flask besitzt und baut darauf auf, da es eine Flask-Unterklasse ist. Es sind bereits viele Funktionen, die von APIs benötigt werden, integriert. Dabei liegt das Hauptaugenmerk von EVE auf der Erzeugung einer API, die so RESTful wie möglich gestaltet ist.

2.13.6 Vergleich und Fazit

Nachdem einige Frameworks kurz erläutert wurden, sollen nun deren Vor- und Nachteile in Bezug auf den Anwendungsfall dieser Arbeit gegenübergestellt werden. Zunächst werden die groben Unterschiede der verschiedenen Systeme verglichen, um eine erste Auswahl zu treffen.

Wohl einer der wichtigsten Unterschiede zwischen den Frameworks ist die ihnen zu Grunde liegende Programmiersprache. Tabelle 8 zeigt dazu einen Überblick:

Framework	Sprache
Flask	Python
Ruby on Rails	Ruby
Django	Python
Spring	Java
Eve	Python

Tabelle 8: Sprachen der vorgestellten Frameworks

Da im Konzept der Einplatinencomputer Raspberry Pi verwendet wird (siehe Kapitel 3.3) und von Herstellerseite Python als Programmiersprache empfohlen wird (Raspberry Pi, 2017), finden sich auch schnell die entsprechenden Python-Bibliotheken zur Verwendung der entsprechenden Sensoren. Es ist zwar durchaus möglich Ruby oder Java auf dem Raspberry Pi zu verwenden, jedoch konnten keine maßgebenden Vorteile dazu gefunden werden.

Die Auswahl konnte so auf die drei Python Frameworks Django, Flask und Eve eingeschränkt werden, die nun etwas genauer betrachtet und verglichen werden:

- Django kommt wie in Abschnitt 2.13.3 beschrieben mit einer sehr großen Auswahl an Funktionen. Im Gegensatz zu Flask besitzt es auch ohne Erweiterungen die Grundvoraussetzungen, um als Framework in dieser Arbeit verwendet zu werden. Django besitzt allerdings auch einige sehr komplexe Fähigkeiten, die nicht benötigt werden und stellt sich so im Vergleich zu Eve als unnötig mächtig dar.

- Flask kommt als schlankes Micro Framework (siehe Abschnitt 2.13.1), das dadurch jedoch auf das Implementieren von Erweiterungen durch den Entwickler setzt. Im Vergleich zu Django oder Eve benötigt Flask also deutlich mehr Aufwand in der Gestaltung der API.

- Eve stellt sich nach Betrachtung der hier vorgestellten Systeme für den vorliegenden Anwendungsfall als guter Mittelweg zwischen Django und Flask dar. Es beinhaltet nahezu alle Funktionen, die die API im vorliegenden Anwendungsfall benötigt und verweist explizit auf die Erweiterung zum Wechsel seiner Standarddatenbank MongoDB zum gewünschten MySQL-Datenbanksystem.

Nach Betrachtung der hier vorgestellten Frameworks konnten diese miteinander zunächst grob verglichen werden. Unter Einbezug der Anforderungen des Anwendungsfalls dieser Arbeit wurden anschließend die Qualitäten der Frameworks abgewogen. So kommt man zu dem Schluss, dass das Python REST API Framework Eve die geeignetste Wahl ist, um die API für den Anwendungsfall in Kapitel 3.1 und das Konzept in Kapitel 3.3 zu erstellen.

2.14 Docker

Docker ist die weltweit führende Software-Container-Plattform. In der Entwicklung soll es rechnersystemabhängige Fehler bei der Zusammenarbeit an Code eliminieren. Im Anwendungsbereich erhöht es die Rechendichte durch die Fähigkeit Anwendungen Seite an Seite in isolierten Containern laufen zu lassen. Weiter ermöglicht Docker Unternehmen die schnelle Auslieferung von Erweiterungen bestehender Systeme (Docker, 2017).

Ein Container-Image ist ein ressourcenschonendes, Stand-Alone-Softwarepaket in dem alles enthalten ist, um eine gewünschte Anwendung auszuführen. Es beinhaltet Dinge wie Code, Laufzeit, Systemwerkzeuge, Systembibliotheken und Einstellungen. Ein großer Vorteil dieser Container Software ist, dass sie den Anwendungen, unabhängig von der tatsächlichen Computerumgebung, eine virtuelle gleichbleibende Umgebung bietet. Dabei isolieren die Container die Anwendungssoftware.

Mehrere Docker-Container, die auf einem Rechner laufen, teilen sich dessen Betriebssystemkern. Dadurch können sie schneller starten und benötigen weniger Rechenleistung und Arbeitsspeicher. Die Images werden aus Dateisystemschichten erstellt und teilen sich gemeinsame Dateien. So kann Speicherplatz und Downloadzeit der Images gespart werden.

Durch die Isolierung der Anwendungen voneinander und von der zugrundeliegenden Infrastruktur bietet Docker standardmäßig bereits Sicherheit durch Isolation. So beschränken sich Probleme einer Anwendung auf den Container in, dem diese ausgeführt wird. Die anderen Container und das Hauptsystem sind in einem solchen Fall vor der fehlerhaften Anwendung geschützt (Docker, 2017).

Abbildung 20 zeigt den schematischen Aufbau eines Docker-Systems mit den unterschiedlichen Containern.

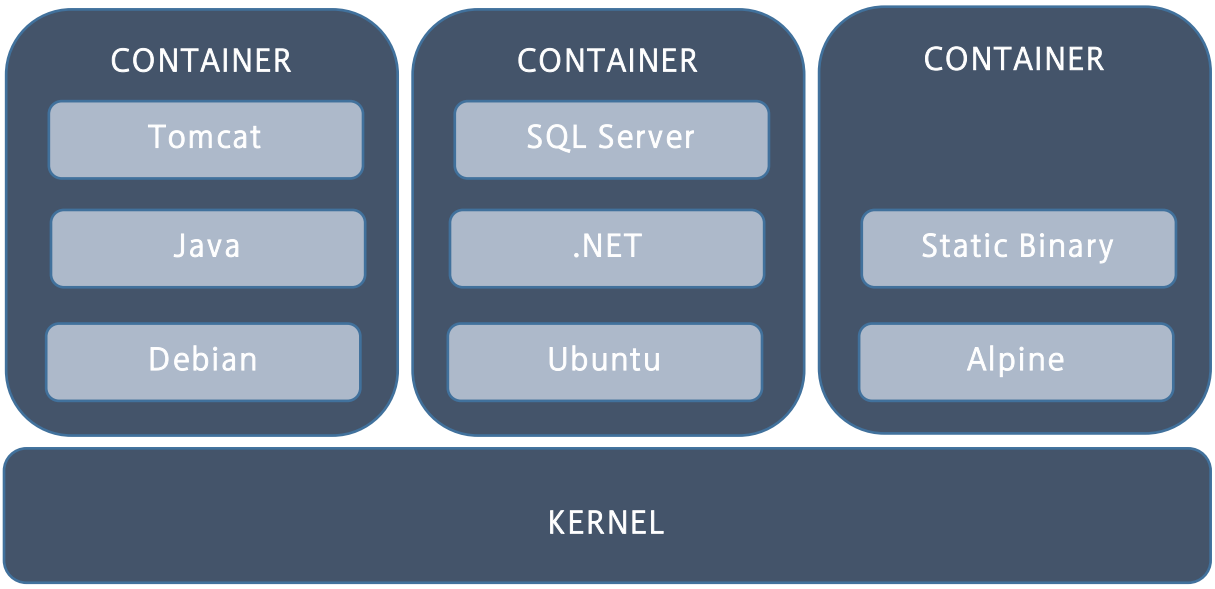

Abbildung 20: Docker–Schema Beispiel (Docker, 2017)

Container und virtuelle Maschinen haben gleichartige Ziele und bieten ähnliche Funktionen. Um den Unterschied zwischen Dockers Container-System und virtuellen Maschinen zu verdeutlichen, soll hier kurz erklärt werden wie diese funktionieren.

Eine **virtuelle Maschine** (VM) ist ein Programm, das als virtueller Computer agiert. Das Betriebssystem des Computers, auf dem die VM ausgeführt wird, stellt dabei das Host Operating System (OS) dar. Analog ist das Betriebssystem, das innerhalb der VM läuft das Guest OS. Getrennt sind diese beiden Betriebssysteme durch den Hypervisor oder Virtual-Machine-Monitor. Dieser dient als abstrahierende Schicht zwischen realer Hardware beziehungsweise des eventuell darauf installierten Host OS und dem Guest OS. Dazu wird eine virtuelle Umgebung, bestehend aus der benötigten Hardware, geschaffen. Durch den Hypervisor können auf einem Computer mehrere VMs laufen. Jede dieser VMs benötigt eine Kopie eines Betriebssystems und die entsprechenden Bibliotheken und Anwendungen (siehe Abbildung 21).

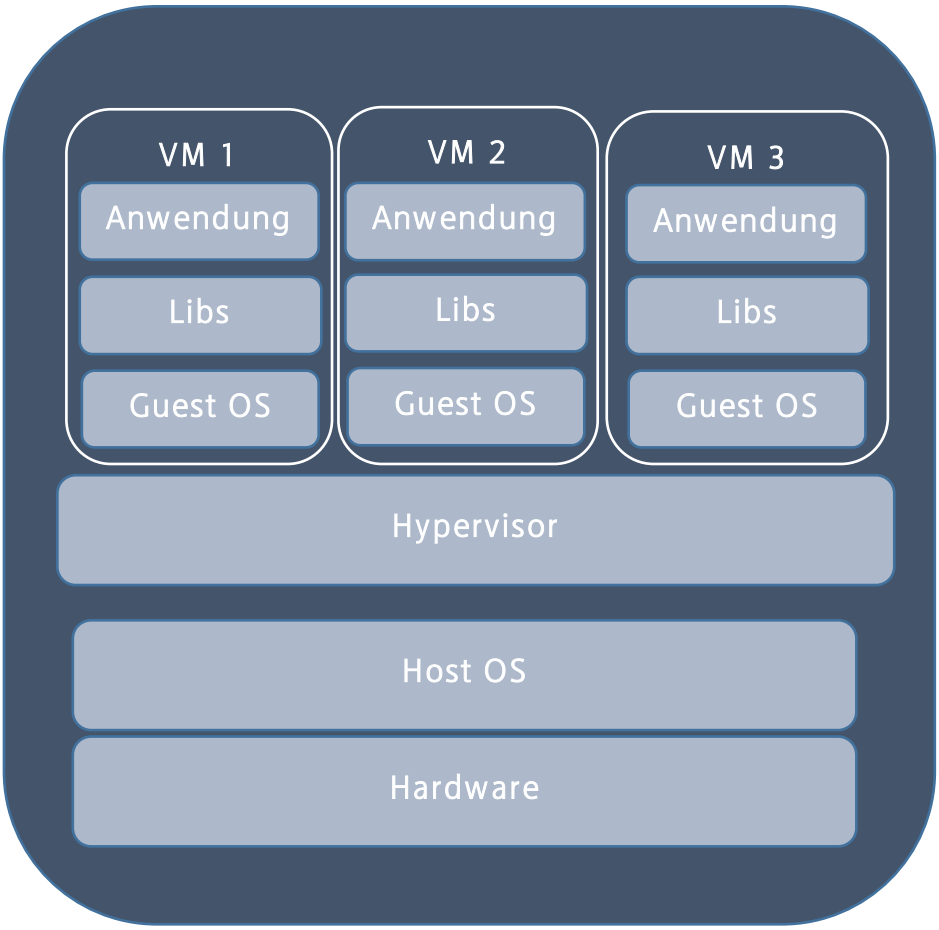

Abbildung 21: Schema virtuelle Maschine

Container sind Abstraktionen auf der Anwendungsebene. Dabei werden Anwendungen und deren Abhängigkeiten zusammen in ein System gefasst (siehe Abbildung 22). Mehrere Container laufen auf dem gleichen Computer und teilen sich einen Betriebssystemkern. Trotzdem laufen die Container als isolierte Prozesse. Da Container jedoch nicht die Hardware, sondern lediglich das Betriebssystem virtualisieren sind sie oft effizienter und Plattformunabhängiger als VMs (Docker, 2017).

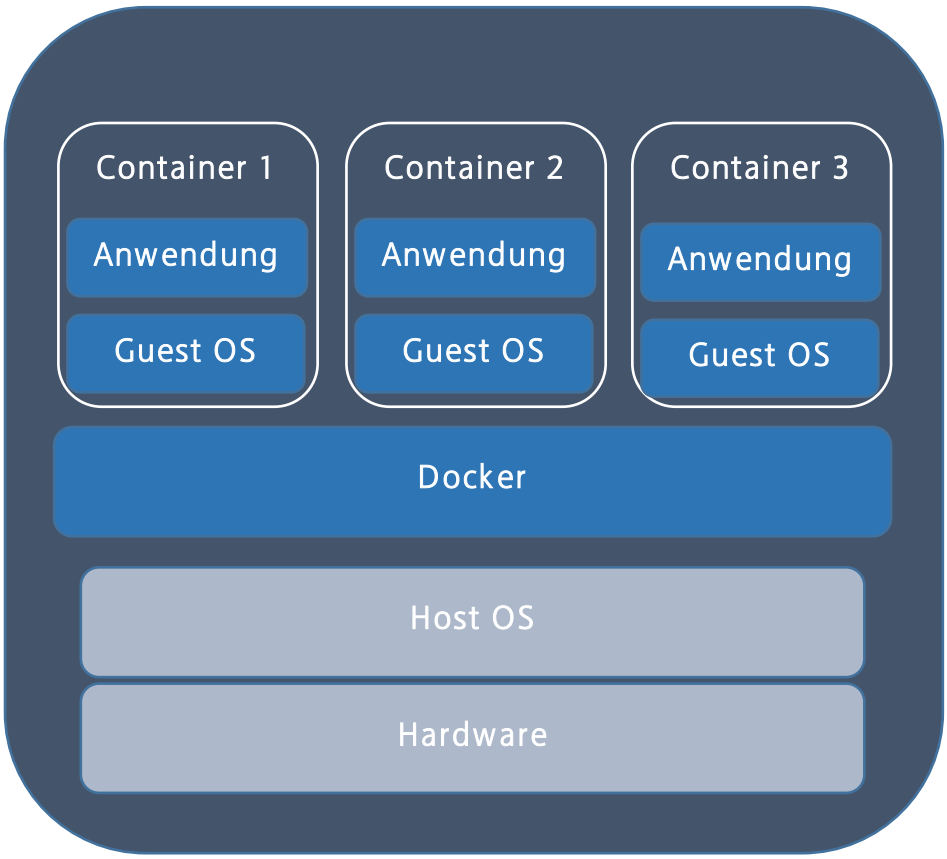

Abbildung 22: Schema Docker-Container

2.14.1 Docker-Compose

Compose dient der Organisation und Ausführung von multi-container Anwendungen. Die Konfiguration einer solchen Anwendung und deren Services regelt eine YAML-Datei, standardmäßig „docker-compose.yml". Dort kann eine Vielzahl von Einstellungen vorgenommen werden, die für die Multi-Container-Anwendung nötig sind – zum Beispiel Einstellungen zu Vernetzung, Startverhalten, Abhängigkeiten oder Ressourcenzuteilung der verschiedenen Container. Ein einzelner Befehl erzeugt dann entsprechend dieser Konfigurationsdatei alle Services.

3 Praxis

Nachdem die nötige Theorie in Kapitel 2 erläutert wurde, soll nun der Einsatz der vorgestellten Systeme in praktischer Anwendung folgen.

3.1 Anwendungsfall

Abbildung 23 zeigt den Aufbau des Servers und der Datenbank mittels Docker. In Container 1 befindet sich die Serverstruktur. Diese läuft auf Ubuntu als Guest OS. Dieser Container beherbergt außerdem die Komponenten der Serverstruktur, bestehend aus Nginx, uWSGI und der Eve-API.

Basis für Container 2, der die MySQL-Datenbank beinhaltet, ist der offizielle MySQL-Container. Dieser kann per Docker-Pull-Request geladen werden. Das Guest OS in diesem Container ist Oracle Linux.

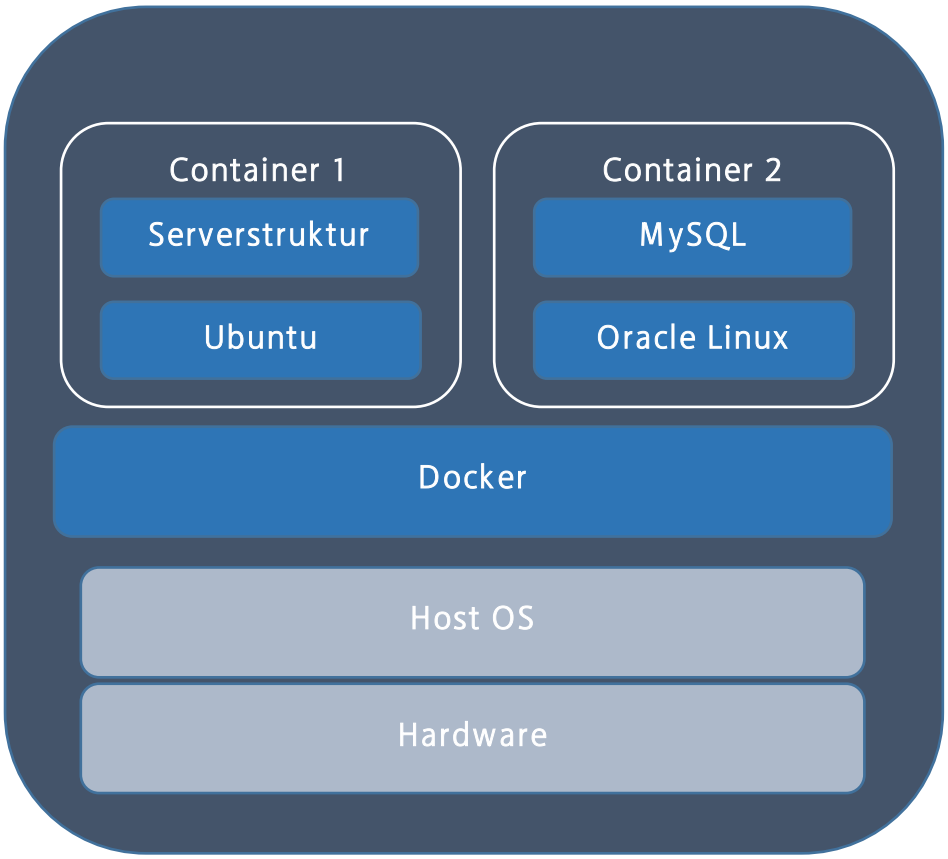

Abbildung 23: Docker-Container der Anwendung

In Abbildung 24 ist der Aufbau der Serverstruktur und der Kommunikationsweg dargestellt. Der Client kontaktiert den HTTP-Server (Nginx), dieser ruft über ein WSGI (uWSGI) die API (Eve-API) auf. Die API ermöglicht nach entsprechender Nutzerauthentifikation und Rechteüberprüfung die Verwaltung der Daten in der Datenbank (MySQL) und sendet die zur Anfrage gehörende Antwort auf gleichem Weg an den Nutzer zurück.

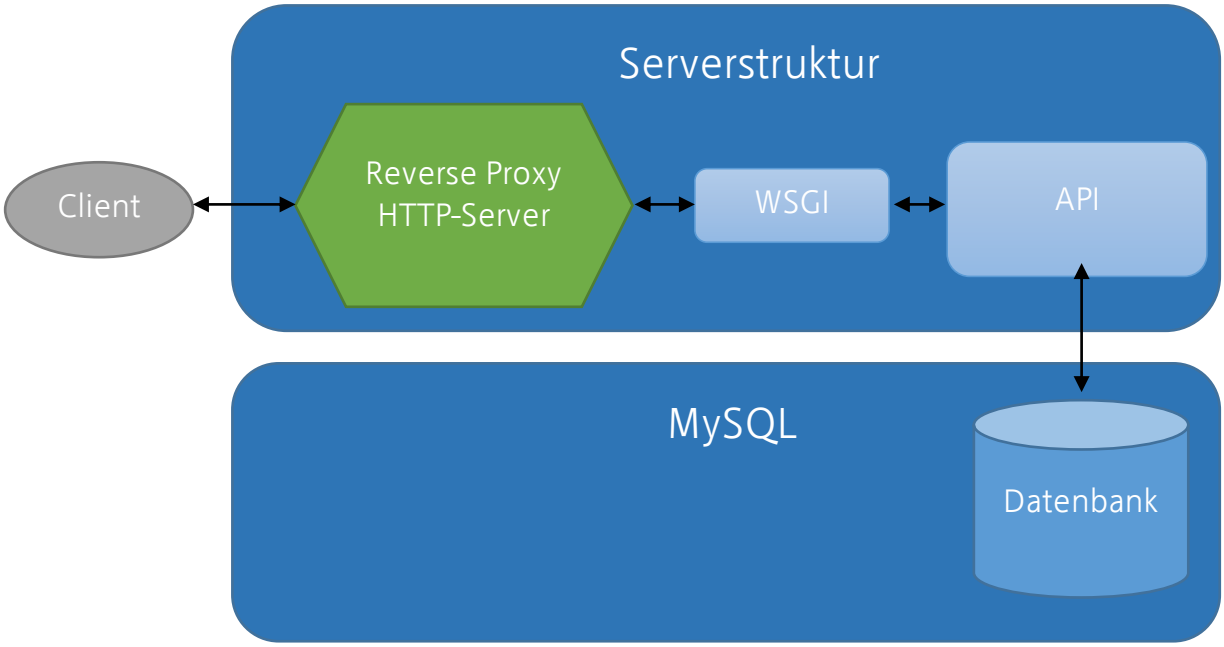

Abbildung 24: Aufbau und Kommunikationsweg

Da sich, zwecks Skalierbarkeit, Kompatibilität und Effizienz, die Datenbank in einem anderen Container als die Serverstruktur befindet, wird Docker-Compose verwendet. Compose ermöglicht die Installation, entsprechende Verknüpfung und Ausführung von mehreren Docker Containern mit nur einem Befehl:

```
sudo docker-compose up
```

Code 14: Docker-Compose Startbefehl

Um die Container im detached-Modus zu starten, ist aber auch das Folgende möglich:

```
sudo docker-compose up -d
```

Code 15: Docker-Compose detached Startbefehl

Mit Docker können mehrere Container auf einem Kernel betrieben werden. So können im Vergleich zu einfachen virtuellen Maschinen Ressourcen gespart werden (siehe Kapitel 2.14).

3.1.1 Nginx Konfiguration

Nginx dient hier als HTTP-Front-End Reverse Proxy Webserver. Die HTTP-Anfragen des Client an den Server werden zunächst von Nginx verarbeitet und mittels entsprechendem Protokoll (hier uwsgi) weitergeleitet (siehe Abbildung 25).

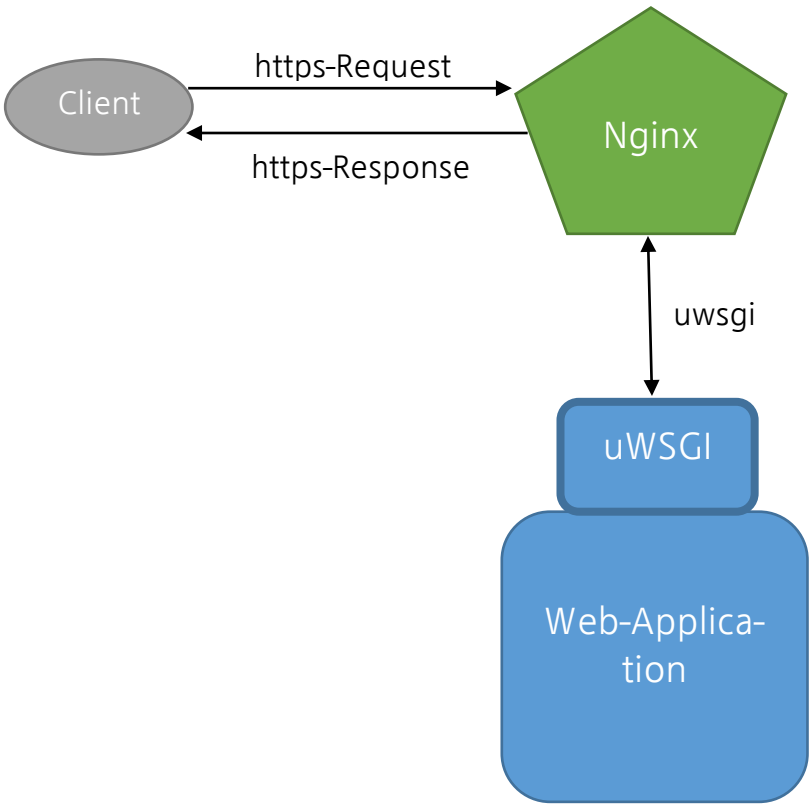

Abbildung 25: Nginx

Der Nginx Server bietet die Möglichkeit einer verschlüsselten Verbindung zum Client mittels https (HyperText Transfer Protocol Secure) (siehe Abschnitt 3.1.2). Trotzdem kann der uWSGI-Server, der die Web-Application bedient, das schnellere, unverschlüsselte uwsgi-Protokoll nutzen, um die weitergeleiten Anfragen zu verarbeiten.

3.1.2 SSL-Einbindung

Um einen sicheren Datentransfer zwischen Client und Server zu gewährleisten, wird das hybride Verschlüsselungsprotokoll SSL (Secure Sockets Layer) genutzt. Für das self-signed SSL Zertifikat wurden dazu mittels Openssl sowohl der benötigte SSL-Schlüssel als auch das SSL-Zertifikat erstellt.

Code 16 in der Nginx Serverkonfiguration ermöglicht Nginx anschließend auf die Dateien zuzugreifen.

```
ssl_certificate /eveapi/ssl/cacert.pem;          # Pfad zu cacert.pem
ssl_certificate_key /eveapi/ssl/privkey.pem;     # Pfad zu privkey.pem
```

Code 16: SSL Schlüssel und Zertifikat für Nginx aus Datei „/eveapi/evenginx“

3.1.3 Docker Deployment von Nginx

Um den Nginx Server richtig einzurichten, genügt es nicht ihn mittels Docker zu installieren und zu starten, es muss auch die Konfigurationsdatei im entsprechenden Verzeichnis ersetzt werden (siehe Code 17).

Nginx Konfigurationsdatei in Docker Container ersetzen:

```
# nginx-Einstellungen importieren

RUN rm /etc/nginx/sites-available/default
RUN rm /etc/nginx/sites-enabled/default
COPY evenginx /etc/nginx/sites-available/default
COPY evenginx /etc/nginx/sites-enabled/default
```

Code 17: Nginx Konfigurationsdatei ersetzen in „/eveapi/dockerfile“

3.1.4 uWSGI

Der uWSGI-Server erlaubt die Verwendung des uwsgi-Protokolls bei der Kommunikation zwischen Nginx und der API. Der uWSGI-Server dient als Web Server Gateway Interface (WSGI). Er stellt damit die Schnittstelle zwischen dem Nginx HTTP-Server und der Python-Anwendung Eve-API dar.

3.1.5 Eve-API

Eve wird als Python REST API Framework zum Erstellen der RESTful Web API genutzt. Die hierbei standardmäßig unterstützte Datenbank ist das, im Vergleich zu MySQL, wenig verwendete MongoDB-Datenbanksystem. Um stattdessen mit MySQL arbeiten zu können, wurde mit den Erweiterungen Eve-SQLAlchemy und SQLAlchemy gearbeitet (siehe 3.1.5.1 MySQL Anbindung).

3.1.5.1 MySQL Anbindung

Die Verbindung zu einer Datenbank, hier MySQL, wird bei Eve in der „settings.py"-Datei geregelt (siehe Code 18).

```
from getsql import sqlpass, dbname


SQLALCHEMY_DATABASE_URI = 'mysql+cymysql://root:'+sqlpass+'@mysql-
server:3306/'+dbname
```

Code 18: MySQL Verbindung aus „/eveapi/settings.py"

Das Passwort für den MySQL Server und der Name der Datenbank werden beim starten des Docker-Containers festgelegt und können in der „docker-compose.yml" Datei gesetzt werden (siehe Code 19).

```
services:
   db:
     image: mysql:5.7
     container_name: mysql-server
     restart: always
     environment:
       MYSQL_ROOT_PASSWORD: secretsqlpassword       # Root-passwort für
MYSQL kann hier gesetzt/geändert werden
       MYSQL_DATABASE: databasename                 # Name der MYSQL-DB
kann hier gesetzt/geändert werden
```

Code 19: MySQL Server Einstellungen in "docker-compose.yml"

3.1.5.2 Authentifikation

Um später den Zugriff auf die Datenbank regeln zu können, bedarf es einer Authentifikation. Hier ist diese durch Token-Authentification gegeben. Dazu wurde zunächst eine Klasse „User“ angelegt, mittels der die Benutzer gespeichert werden können (siehe Code 20).

```
class User(CommonColumns):
    __tablename__ = 'users'
    _id = Column(Integer, autoincrement=True)
    login = Column(String(140), primary_key=True)
    password = Column(String(140))
    roles = Column(String(140))
```

Code 20: User Klasse aus "/eveapi/webapp/models/UserD.py"

Der Authentifikationsprozess wird in dieser Datenbank nach den vom Benutzer an den Login-Endpunkt („/login“) übermittelten Zugangsdaten („login“ und „passwort“) suchen. Falls der Nutzer authentifiziert werden konnte, wird ein zeitlich beschränkt gültiger Token generiert. Dazu musste die von Eve standardmäßig genutzte Methode „TokenAuth“ angepasst werden. Die hierzu nötigen Funktionen sind ebenfalls in der User-Klasse festgelegt.

Ein erfolgreicher Login-Vorgang mittels Curl kann dann wie folgt aussehen:

```
Anfrage:
curl -k -H "Content-Type: application/json"
        -X POST -d '{"username":Testname", "password":"Testpasswort"}'
        https://172.18.0.3/login

Antwort:
{
    "token": „eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODk
              IiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2c
              ab30RMHrHDcEfxjoYZgeFONFh7HgQ"
}
```

Code 21: Curl Login Beispiel

Wenn der User in der Datenbank gefunden wurde, wird ein Token generiert und an den Client zurückgesendet. Dieser Token dient anschließend für eine beschränkte Zeit der Authentifikation, um geschützte Endpunkte zu erreichen (siehe Code 22).

```
Anfrage:
curl -k -H "Authorization: TokenAuth
        „eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkIiwibm
        FtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cab30RMHrHDcEfxj
        oYZgeFONFh7HgQ"
        -X GET
        https://172.18.0.3/Endpunkt
```

Code 22: Zugriff auf geschützten Endpunkt mit Token

3.1.5.3 Zugriffsrechte

Wie in Code 20 zu sehen ist, kann jedem User neben Nutzername und Passwort auch eine Rolle zugewiesen werden. Eve verfügt über die Möglichkeit Endpunkte mit bestimmten Rechten für bestimmte Rollen zu versehen. Diese Rechte werden in der „settings.py" über den DOMAIN-Dictionary den entsprechenden Endpunkten zugewiesen.

Um eine möglichst einfache und übersichtliche Nutzung dieser Rechtevergabe zu gewährleisten, wurde dazu eine eigene Datei „accessrights.py" erstellt. Diese Datei dient ausschließlich der Endpunktsicherung. In Code 23 ist das Schema zur Endpunktsicherung durch unterschiedliche Rechte und Rollen zu sehen.

Python Eve bietet eine Vielzahl von weiteren Endpunktkonfigurationen, die unter http://python-eve.org/config.html#local zu finden sind. Die im Zuge dieser Arbeit benötigten Konfigurationen sind in Code 23 dargestellt:

```
1    # Zugriff ohne Authentifikation
2    DOMAIN['Endpunkt1'].update({
3        'authentication': None,
4        'resource_methods': ['POST', 'GET']
5    })
6
7    # Zugriffsrechte nur für bestimmte Rollen
8    DOMAIN['Endpunkt2'].update({
9        'allowed_roles': ['admin', 'superuser'],
10       'resource_methods': ['GET', 'POST', 'DELETE']
11   })
12
13   # Unterschiedliche Operationsrechte für unterschiedliche Rollen
14   DOMAIN['Endpunkt3'].update({
15       'allowed_write_roles': ['admin', 'superuser'],
16       'allowed_read_roles': ['user'],
17       'resource_methods': ['GET', 'POST', 'DELETE']
18   })
19
```

Code 23: Beispiel zur Endpunktsicherung in "/eveapi/webapp/accessrights.py"

Standardmäßig sind alle über den DOMAIN-Dictionary festgelegten Endpunkte der API durch Authentifikation geschützt. In Code 23 Zeile 3 wird der Zugriff auf den Endpunkt „Endpunkt1“ ohne Authentifikation ermöglicht.

Um weitere Methoden neben GET zuzulassen, müssen diese explizit erlaubt werden. (zu sehen in den Zeilen 4, 10 und 17)

Zeile 9 aus Code 23 beschränkt den Zugriff auf Endpunkt2 auf bestimmte Rollen. Mittels der Ressourcenendpunktkonfiguration „allowed_roles“ wird jeglicher Zugriff auf den Endpunkt für bestimmte Rollen beschränkt. Dies geschieht dabei unabhängig von der genutzten Methode.

In den Zeilen 15 und 16 wird der Zugriff methodenabhängig für die entsprechenden Rollen gewährt beziehungsweise beschränkt. „allowed_read_roles“ enthält dabei die Liste aller Nutzerrollen, die Leserechte haben, also GET-Methode anwenden dürfen. „allowed_item_write_roles“ erteilt Rechte für schreibende Tätigkeiten. Diese bestehen aus den Methoden: PUT, PATH und DELETE. Voraussetzung ist dabei immer, dass die entsprechende Operation auch durch die „ressource_methods“ Liste generell genehmigt ist.

3.1.5.4 Erstellen und Verwalten der Tabellen

Auch die Tabellen werden bei Eve normalerweise von der „settings.py“-Datei verwaltet. Um das Erstellen und Verwalten der Tabellen übersichtlicher und einfacher zu gestalten, wurde diese Aufgabe in einer eigenen Datei „tables.py“ realisiert. Die Tabellen werden als Klassen eingefügt und können mittels SQLAlchemy für MySQL statt für MongoDB generiert werden (siehe Code 24).

```
class Beispieltabelle(CommonColumns):
    __tablename__ = 'Beispielname'
    Spaltenname1 = Column(Integer, primary_key=True, autoincrement=True)
    Spaltenname2 = Column(String(20))
    Spaltenname3 = Column(Integer())
```

Code 24: Beispiele für Tabellen aus "/eveapi/tables.py"

Um die so erstellten Tabellen voll nutzen zu können, müssen sie dem DOMAIN Dictionary zugeführt werden. Dieser wird anschließende durch die entsprechenden Rechte für die einzelnen Tabellen in „accessrights.py“ erweitert, bevor er in der „settings.py“ geladen werden kann. Normalerweise erstellt Eve den DOMAIN-Dictionary aus den entsprechenden Eingaben in die „settings.py“ Datei. Da hier jedoch aus Gründen der Übersichtlichkeit und Vereinfachung der API das Erstellen der Tabellen beziehungsweise der Rechte in jeweils eigenen Dateien („tables.py“, beziehungsweise „accessrights.py“) realisiert ist, musste der DOMAIN-Dictionary selbst erstellt werden. Danach kann dieser komplett in „settings.py“ importiert werden.

Zunächst müssen alle Tabellen in „tables.py“ identifiziert werden, um dann eine Liste zu generieren, die alle Tabellen enthält. Anschließend muss der DOMAIN-Dictionary so erstellt werden, dass Eve damit arbeiten kann. In Code 25 wird außerdem die User Klasse aus „eveapi/webapp/models/UserD.py“ dem Dictionary zugeführt, da diese unabhängig von den erstellten Tabellen als Nutzerdatenbank immer existieren soll. Danach können die in Code 24 beschriebenen Tabellen aus der Tabellenliste entsprechend in den Dictionary eingefügt werden.

```
# Erstellen einer Liste der Tabellen für den DOMAIN-Dict

current_module = sys.modules[__name__]


def tabellen():
    tabellenliste = inspect.getmembers(sys.modules[__name__], lambda
member: inspect.isclass(member) and member.__module__ == __name__)
    return tabellenliste


# DOMAIN-Dict aus den oben angelegten Tabellen und der User-Tabelle aus
models.UserD die zur Authentifizierung dient
DOMAIN = {}
for j, k in tabellen():
        if not DOMAIN:
            registerSchema('User')(User)
            new_domain = 'User'
            new_schema = User._eve_schema['User']
            DOMAIN[new_domain] = new_schema

        registerSchema('j')(k)
        new_domain = j
        new_schema = k._eve_schema['j']
        DOMAIN[new_domain] = new_schema
```

Code 25: Eigenes Erstellen des DOMAIN Dictionary in "/eveapi/tables.py"

3.2 Workflow

Die gesamte API wurde so gestaltet, dass sie eine möglichst einfache Installation und Konfiguration ermöglicht. Hier soll gezeigt werden, wie diese ablaufen kann.

3.2.1 Voraussetzungen

Voraussetzung zur Inbetriebnahme der API sind Git und Docker-Compose.

Git Installation:

Git zählt zu den meistverwendeten Programmen zur verteilten Versionsverwaltung. Da es gut dokumentiert ist und sowohl unter Linux, Mac als auch Windows läuft, bietet es eine gute Möglichkeit zur Bereitstellung der API.

Für Mac und Windows basierende Systeme gibt es eine offizielle Installationsdatei zur geführten Installation. Diese ist unter https://git-scm.com/downloads zu finden.

Bei Linux führt folgender Befehl im Terminal die Installation von Git aus:

```
sudo apt-get install git
```

Code 26: Git Installationsbefehl

Falls ein anderer Package Manager installiert ist, sind die alternativen Befehle auf https://git-scm.com/download/linux zu finden.

Docker Compose Installation:

Für Docker Compose muss unter Linux zunächst Docker installiert werden. Für Mac und Windows 10 existieren geführte Installationsdateien von Docker CE unter https://www.docker.com/community-edition. Diese enthalten bereits das Docker Compose Tool.

Unter Linux wird zunächst Docker CE benötigt, das mithilfe folgender 5 Befehle installiert werden kann.

```
1    sudo apt-get -y install \
       apt-transport-https \
       ca-certificates \
       curl

2    curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo
     apt-key add -

3    sudo add-apt-repository \
           "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
            $(lsb_release -cs) \
            stable"

4    sudo apt-get update

5    sudo apt-get -y install docker-ce
```

Code 27: Installationsbefehle Docker CE

Zum Prüfen der Installation eignet sich der Hello-World-Befehl:

```
sudo docker run hello-world
```

Code 28: Prüfen der Docker-Installation

Nach erfolgreicher Installation von Docker CE, kann das Docker Compose Tool installiert werden. Der einfachste Weg ist dabei mittels pip:

```
pip install docker-compose
```

Code 29: Docker-Compose Installation

Der Befehl in Code 30 kann zur Überprüfung einer erfolgreichen Installation ausgeführt werden:

```
docker-compose –version
```

Code 30: Überprüfen der Docker-Compose Installation

Alternative Installationswege sind auf https://docs.docker.com/compose/install/ zu finden.

Wie die nötigen Installationen gezeigt haben, besteht eine Kompatibilität zu den drei größten Betriebssystem Linux, Mac und Windows. Einschränkungen in den einzelnen Systemen sind dabei durch Docker gegeben. Folgende Versionen der Betriebssysteme werden unterstützt:

- Windows 10
- Apple Mac OS Yosemite 10.10.3 oder höher
- Yakkety 16.10 (64-bit)
- Xenial 16.04 (LTS) (64-bit)
- Trusty 14.04 (LTS) (64-bit)

3.2.2 Klonen des Git Repository

Klonen des Git Repository bedeutet in diesem Fall nichts anderes als den Code der Anwendung von einem Server auf den lokalen Computer zu laden. Dies geschieht mittels Code 31.

```
sudo git clone https://OliWi@bitbucket.org/OliWi/docker-eve.git
```

Code 31: Klonen des Git Repository

Dabei wird das komplette Repository automatisch auf dem System im Ordner „docker-eve“ gespeichert (das Repository ist im Moment noch nicht öffentlich zugänglich).

3.2.3 Erstellen der Tabellen

SQLAlchemy erlaubt die Nutzung einer SQL-Datenbank anstatt der von Eve vorgesehenen MongoDB. Somit kann ein SQL-Datenmodell erstellt oder ein bereits bestehendes weiter verwendet werden.

Die gewünschten Tabellen können in „docker-eve/tables.py“ im entsprechenden Feld eingefügt werden. Als Beispiel wird das Übertragen der folgenden zwei SQL-Tabellen gezeigt:

Mitarbeiter
_id (primary key)
name
id_person

Tabelle 9: Beispiel Mitarbeitertabelle

Abteilung
_id (primary key)
name
id_abteilung

Tabelle 10: Beispiel Abteilungstabelle

Code 32 zeigt, wie Tabelle 9 und Tabelle 10, die typisches SQL-Format haben, in „tables.py“ eingefügt werden können. Die API übernimmt anschließend die Übertragung an die MySQL Datenbank.

```
class Mitarbeiter(CommonColumns):
    __tablename__ = 'Mitarbeiter'
    _id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String(20))
    id_person = Column(Integer())


class Abteilung(CommonColumns):
    __tablename__ = 'Abteilung'
    _id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String(20))
    id_abteilung = Column(Integer())
```

Code 32: Umsetzung von SQL-Tabellen

3.2.4 Festlegen der Rechte

Die in Kapitel 3.2.3 erstellten Tabellen können in „docker-eve/accessrights.py“ mit Rechten versehen werden. Die Form der gebräuchlichsten Rechte ist im Beispiel von Code 33 gezeigt. Standardmäßig sind alle über den DOMAIN Dictionary festgelegten Endpunkte der API durch Authentifikation geschützt. In Code 33 Zeile 3 wurde zu Demonstrationszwecken der Zugriff auf den „User“-Endpunkt ohne Authentifikation ermöglicht.

```
1   # Zugriff ohne Authentifikation
2   DOMAIN['User'].update({
3       'authentication': None,
4       'resource_methods': ['POST', 'GET']
5   })
6
7   # Zugriffsrechte nur für bestimmte Rollen
8   DOMAIN['Mitarbeiter'].update({
9       'allowed_roles': ['admin', 'superuser'],
10      'resource_methods': ['GET', 'POST', 'DELETE']
11  })
12
13  # Unterschiedliche Operationsrechte für unterschiedliche Rollen
14  DOMAIN['Abteilung'].update({
15      'allowed_write_roles': ['admin', 'superuser'],
16      'allowed_read_roles': ['user'],
17      'resource_methods': ['GET', 'POST', 'DELETE']
18  })
19
```

Code 33: Beispiel Rechtevergabe der erstellten Tabellen

3.2.5 Starten der Docker-Container

Mittels des Befehls in Code 34

Code 34 wird automatisch sowohl der MySQL-Server als auch die API installiert, gestartet und miteinander verbunden.

```
sudo docker-compose up -d
```

Code 34: Docker-Compose start (Workflow)

Es können dann alle definierten Endpunkte (Tabellen) über die URL „https://IP/Endpunktname“ erreicht werden. Falls nicht anders in den Rechten festgelegt, sind die Endpunkte durch Authentifikation (siehe Abschnitt 3.1.5.2) geschützt.

Die IP des Docker Containers kann mittels Code 35 gefunden werden. Diese zeigt sich dann in der Ausgabe der NetworkSettings, wie in Abbildung 26 unter dem Punkt „IPAddress“ zu sehen ist. Docker bietet zwar die Möglichkeit Containern feste IP-Adressen zuzuweisen, dies würde jedoch die Chancen des Auftretens von IP-Konflikten erhöhen.

```
sudo docker inspect eve-api
```

Code 35: IP des Docker-Containers finden

Abbildung 26: docker inspect Ausgabe

3.3 Beispielhafte Umsetzung – Konzept

Hier soll eine Anwendungsmöglichkeit der API gezeigt werden. Da der Raspberry Pi zu den meistverkauften Einplatinencomputern zählt (MagPi, 2017), wurde dieser als Basiselement gewählt. Dieser arbeitet mit einer anderen CPU Architektur, nämlich ARM statt dem üblichen x86/x64 Intel oder AMD. Deshalb mussten geringfügige Anpassungen vorgenommen werden. So musste ein anderes Guest OS Image gewählt werden (Resin Rpi-Raspbian anstatt Ubuntu). Außerdem muss der Docker Container auf die GPIO (General Purpose Input/Output) Pins des Raspberry Pi zugreifen können. Dies konnte durch die entsprechende Gestaltung der Container-Startbedingungen in der „docker-compose.yml" Datei erreicht werden (Siehe Code 37)

Als Beispiel soll gezeigt werden, wie ein Python Skript, das über die GPIO Pins des Raspberry Pi die Daten eines Temperatur- und Luftfeuchtigkeitssensors abfragt, implementiert wird.

Nach Implementierung der API, sollen die Daten des Sensors in einer MySQL Datenbank gespeichert und durch die API geschützt und verwaltet werden. Um die Skalierbarkeit zu zeigen, wurden weitere Bedienungsmöglichkeiten des Sensorprogramms durch die API geschaffen. So soll das Programm zur Datenerfassung auch durch die API gesteuert werden. Die Steuerung besteht hier aus dem Starten und Beenden der Sensorabfrage sowie einer Zustandsüberprüfung, die Rückmeldung gibt, ob das entsprechende Programm läuft, also momentan Daten erfasst.

3.3.1 Aufbau

In Abbildung 27 ist der Gesamtaufbau und die Verbindung der Docker Container gezeigt. Das Programm zur Datenerfassung wurde mit einer einfachen Eve-API ausgestattet, und kann so in einem eigenen Docker-Container („temp-api") laufen. Docker bietet die Möglichkeit, bei Bedarf auch weitere Programme in eigenen Containern anzubinden. Man erkennt schnell die einfache Möglichkeit der Erweiterung und damit die gute Skalierbarkeit dieses Systems.

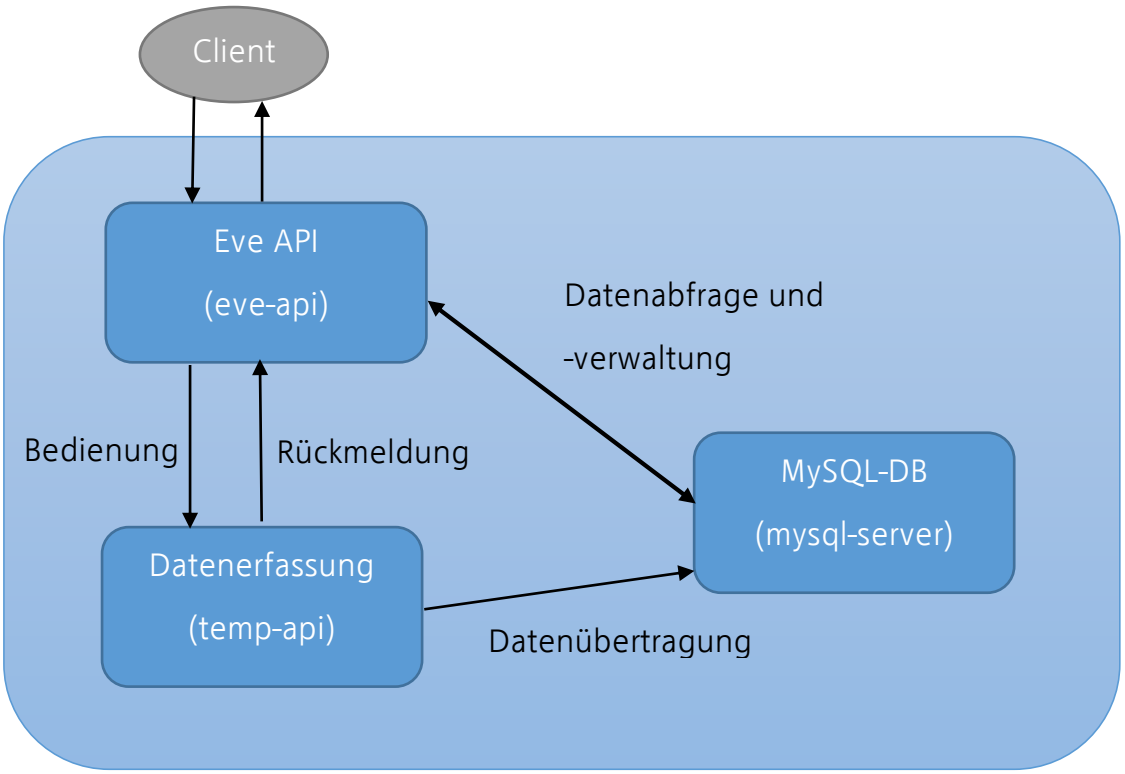

Abbildung 27: Konzeptaufbau

3.3.2 Anbindung der Datenerfassung an das bisherige System

Die Datenerfassung steht hier stellvertretend für ein beliebiges Programm in Python, daher wird auf deren Funktionsweise nicht genauer eingegangen. Sie dient hier lediglich als Beispiel zur Einbindungsmöglichkeit von Anwendungen an die API.

Zunächst wurde ein Dockerfile erstellt, um den Container zu generieren in dem die Datenerfassung laufen soll. In diesem Container werden alle, für das Programm der Datenerfassung, nötigen Pakete installiert (Siehe Code 36).

```
FROM resin/rpi-raspbian:latest

RUN apt-get update && apt-get install -y \
     build-essential \
     python3 \
     python3-dev \
     libmysqlclient-dev \
     python3-pip \
     python3-rpi.gpio

COPY . /tempapi

WORKDIR /tempapi

RUN pip3 install eve \
     mysqlclient \
     numpy

EXPOSE 5000

CMD python3 /tempapi/evepi.py
```

Code 36: „/docker-eve/tempapi/dockerfile“

Anschließend wurde das bestehende „/docker-eve/docker-compose.yml“ File um Code 37 erweitert, sodass beim üblichen Start der API und der MySQL Datenbank durch „docker-compose up“ auch der Container der Datenerfassung mit den entsprechenden Konfigurationen gestartet wird.

```
tempapi:
  container_name: temp-api
  build: ./tempapi
  image: piimg
  ports:
   - "5000:5000"
  devices:
   - "/dev/mem:/dev/mem"
   - "/dev/ttyAMA0: /dev/ttyAMA0"
  privileged: true
```

Code 37: Datenerfassung in Docker-Compose aus „/docker-eve/docker-compose.yml“

3.3.3 Datenübertragung zur MySQL Datenbank

Die Übertragung der Daten zur MySQL-Datenbank, die sich im „mysql-server“- Container befindet, wurde mittels MySQLdb realisiert und läuft nahezu komplett nach MySQLdb-Dokumentation ab. Einzige Ausnahme ist die Host-Adresse des MySQL Servers. Hier kann der jeweilige Containername genutzt werden („mysql-server“) und Docker ermöglicht die Verbindung unabhängig von der IP-Adresse des Containers (siehe Code 38 unter Host).

```
db = MySQLdb.connect(host="mysql-server", user="root",
passwd="secretsqlpassword", db="databasename")
```

Code 38: MySQL Verbindung aus „/docker-eve/tempapi/mysql.py"

3.3.4 Bedienung der Datenerfassung

Folgende Bedienungsmöglichkeiten für das Programm der Datenerfassung wurden beispielhaft gewählt und implementiert:

- Starten
- Stoppen
- Zustandsüberprüfung (läuft die Datenerfassung oder nicht)

Diese Aufgaben übernimmt eine kleine Eve API (Siehe Code 39) im „temp-api“ Container, die anschließend von der Haupt-Eve-API gesteuert werden soll (Vgl. Abbildung 27).

```
import subprocess
from eve import Eve


app = Eve()

s = None

@app.route('/start')
def start1():
    global s
    s = subprocess.Popen(['python3', 'temp.py'], stdout=subprocess.PIPE)
    return('gestartet')

@app.route('/stop')
def stop1():
    if s is None:
        return('not running')
    else:
        s.terminate()
        s.wait(timeout=5)
        return('gestoppt')

@app.route('/check')
def check1():
    if s is None:
        return('not running')
    if s.poll() is None:
        return('running')
    else:
        return('not running')

if __name__=='__main__':
    app.run(host='0.0.0.0')
```

Code 39: Bedienung Datenerfassung „/docker-eve/tempapi/evepi.py"

Damit die Steuerung der Datenerfassung durch Authentifikation und Rechte geschützt werden kann, werden die Endpunkte aus Code 39 dem Nutzer nicht direkt zugänglich gemacht, sondern von der „Haupt-API“ angesteuert. Dank Docker kann hier wieder der Container-Name („temp-api“) unabhängig von der Container-IP genutzt werden (siehe Code 40). Es sei angemerkt, dass die Endpunkte in dieser Form nicht durch Authentifikation geschützt sind, da sie nicht über den DOMAIN-Dictionary generiert werden. Falls gewünscht, lassen sich auch diese Endpunkte über die bereits definierte Authentifikation schützen. Dazu wird lediglich der „ @requires_auth()“ Decorater eingefügt.

```
@app.route('/check')
def checken():
    check = urllib.request.urlopen('http://temp-api:5000/check')
    responsecheck = check.read()
    return (responsecheck)


@app.route('/stop')
def stoppen():
    stop = urllib.request.urlopen('http://temp-api:5000/stop')
    responsestop = stop.read()
    return (responsestop)


@app.route('/start')
def starten():
    start = urllib.request.urlopen('http://temp-api:5000/start')
    responsestart = start.read()
    return (responsestart)
```

Code 40: Custom Endpoints in Eve-API aus „/docker-eve/webapp/__init__.py"

3.3.5 Datenabfrage

Um die Daten aus der MySQL-Datenbank mit allen Funktionen von SQLAlchemy abfragen zu können, wurde die Tabelle 11 in „/docker-eve/tables.py“ eingefügt (siehe Code 41).

temperatur
id (primary key, AUTO_INCREMENT)
Datum
Zeit
Temperatur
Luftfeuchtigkeit

Tabelle 11: Schema Temperaturtabelle

```
class temperatur(CommonColumns):
    __tablename__ = 'temperatur'
    id = Column(Integer, primary_key=True, autoincrement=True)
    DATUM = Column(String(12))
    ZEIT = Column(String(20))
    TEMPERATUR = Column(String(20))
    LUFTFEUCHTIGKEIT = Column(String(20))
```

Code 41: Temperaturtabelle in "/docker-eve/tables.py“

3.3.6 Beispielbedienung

Hier soll die mögliche Bedienung nach dem Starten der Container gezeigt werden.

Zunächst ermittelt man die IP-Adresse des „eve-api“ Containers wie in Abschnitt 3.2.5 beschrieben (diese ist hier „172.18.0.4“), um anschließend die Endpunkte mittels Curl aufrufen zu können.

- Zustandsüberprüfung

```
Anfrage:
curl -k https://172.18.0.4/check

Antwort:
not running
```

Code 42: Curl Check-Endpunkt

Die Zustandsüberprüfung sagt aus, dass die Datenerfassung nicht läuft.

- Start

```
Anfrage:
curl -k https://172.18.0.4/start

Antwort:
gestartet
```

Code 43: Curl Start-Endpunkt

Die Antwort „gestartet" sagt aus, dass die Datenerfassung nun läuft.

Die Zustandsüberprüfung bestätigt dies mit der Rückgabe „running":

```
Anfrage:
curl -k https://172.18.0.4/check

Antwort:
running
```

Code 44: Zustandsüberprüfung "running"

Zu Demonstrationszwecken wird noch der Eintrag in der MySQL Datenbank überprüft. Dieser zeigt folgende Eintragungen:

```
mysql> select * from temperatur;
+----+------------+-----------------+-------------+------------------+
| ID | DATUM      | ZEIT            | TEMPERATUR | LUFTFEUCHTIGKEIT |
+----+------------+-----------------+-------------+------------------+
|  1 | 2017-03-31 | 09:58:51.174846 | 24.0        | 40.0             |
|  2 | 2017-03-31 | 09:59:54.776087 | 24.0        | 39.8             |
|  3 | 2017-03-31 | 10:00:58.319847 | 24.0        | 39.6             |
+----+------------+-----------------+-------------+------------------+
3 rows in set (0.00 sec)

mysql>
```

Abbildung 28: MySQL Datenbankeintrag

Wie man in Abbildung 28 sieht, wurde die Datenerfassung gestartet und die Sensordaten werden nun erfolgreich in die entsprechende Tabelle in der MySQL Datenbank eingetragen.

- Abfrage im Browser

Da die Tabelle auch in der API eingetragen wurde, können nun SQLAlchemy Abfragen ausgeführt werden. Der „temperatur"-Endpunkt liefert uns im Browser die gewünschten Daten im XML-Format:

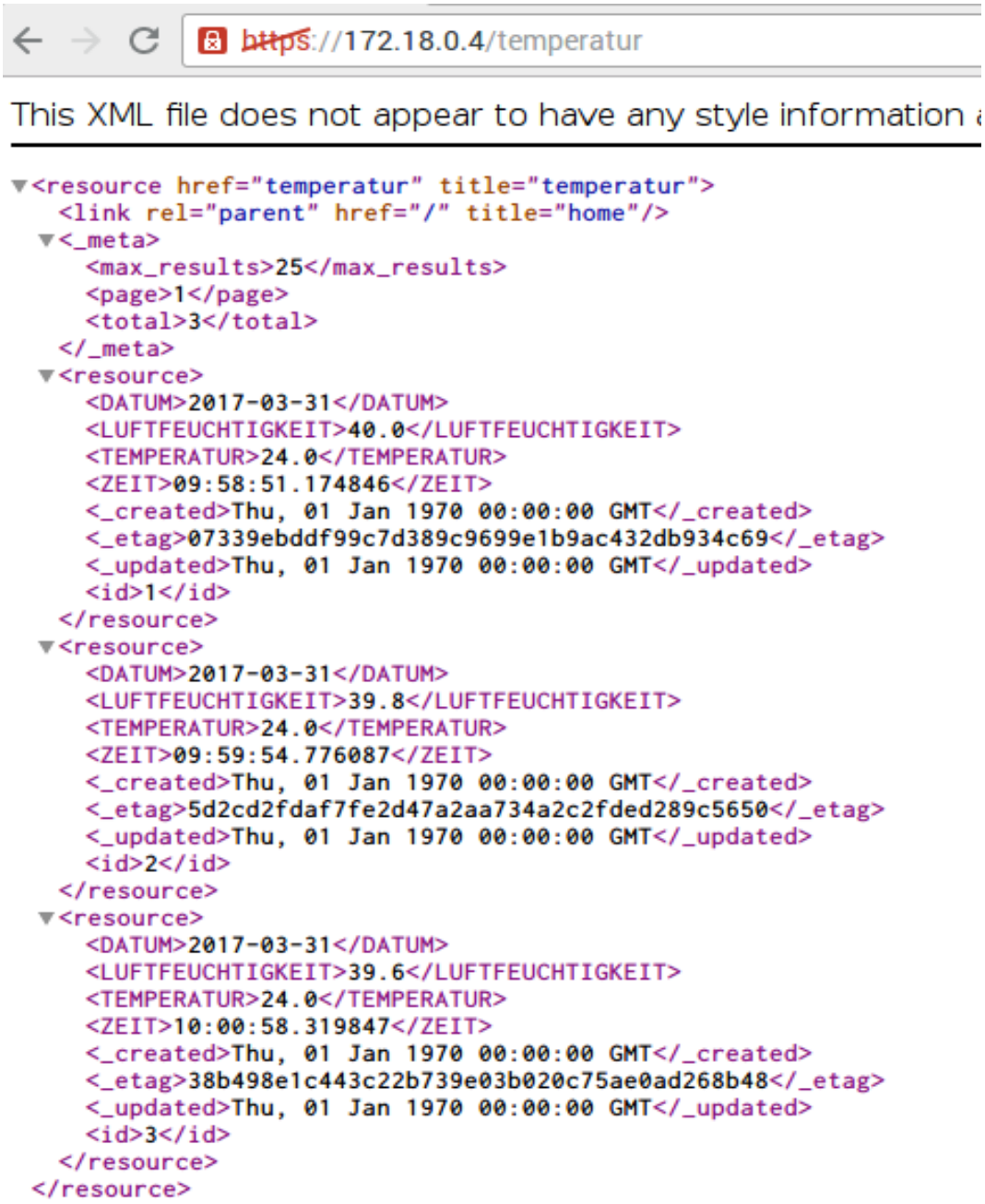

Abbildung 29: Ausgabe im Browser unsortiert

Die Ausgabe kann mittels SQLAlchemy auf verschiedene Arten gesteuert beziehungsweise sortiert werden. Eine Vielzahl von Möglichkeiten findet man dazu unter http://eve-sqlalchemy.readthedocs.io. Als Beispiel zur SQLAlchemy Abfrage soll nun die Ausgabe sortiert werden:

Der neueste Eintrag soll an erster Stelle stehen. Dies gelingt mittels entsprechendem Zusatz am Ende der URL:

- https://172.18.0.4/temperatur?sort=[("id", -1)]

Dieser Ausdruck sortiert die Ausgabe nach dem Feld „id“ absteigend und führt damit zu folgender Ausgabe:

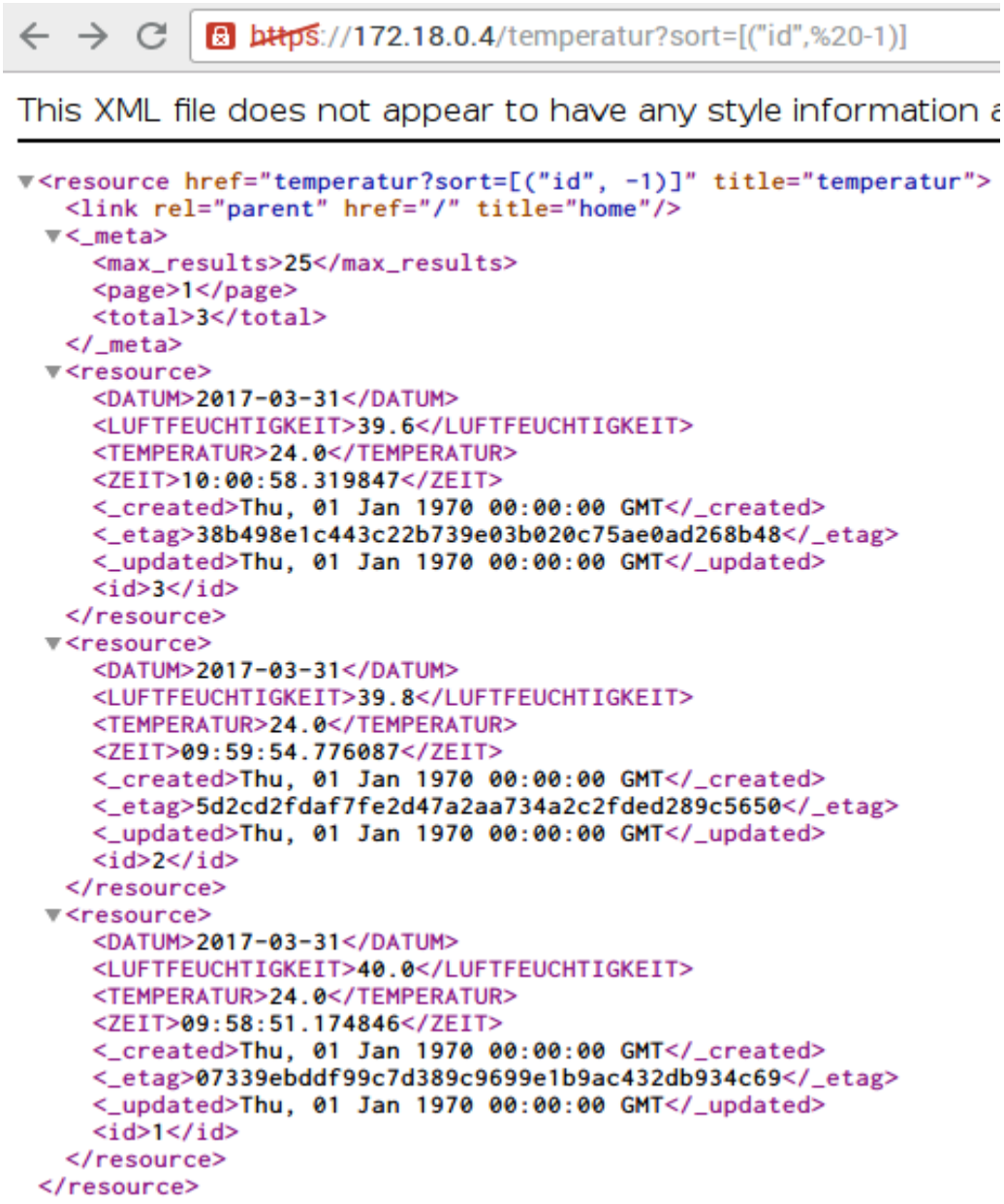

Abbildung 30: Ausgabe im Browser sortiert

- Stoppen

```
Anfrage:
curl –k https://172.18.0.4/stop

Antwort:
gestoppt
```

Code 45: Stopp bei laufender Datenerfassung

Die Rückmeldung „gestoppt" sagt aus, dass die Datenerfassung erfolgreich gestoppt wurde. Die Zustandsüberprüfung bestätigt, dass die Datenerfassung nun nicht mehr läuft:

```
Anfrage:
curl –k https://172.18.0.4/check

Antwort:
not running
```

Code 46: Zustandsüberprüfung nach Stoppen

Es wurde gezeigt, dass sich neben den üblichen Funktionen der Eve API noch schnell weitere Funktionsmöglichkeiten zur Steuerung von zum Beispiel datenerfassenden Programmen implementieren lassen, und wie diese aussehen können.

4 Zusammenfassung

Für den unerfahrenen Nutzer kann der Stand der Technik im Fachbereich der Informationstechnik überwältigend wirken. Es existiert eine Vielzahl an nutzbaren Komponenten oder Modulen. Nicht alle davon sind jedoch notwendig oder vorteilhaft, um eine Systemstruktur zu gestalten, die in ihrer Skalierbar- und Erweiterbarkeit mit den Entwicklungen in diesem Technikbereich mithalten kann.

Die Grundlagentheorie, die in dieser Arbeit verfasst wurde, vermittelt dem Leser nicht nur das nötige Hintergrundwissen, um die Arbeitsweise und den Aufbau der vorgestellten Anwendung zu verstehen, sie zeigt auch den Prozess der Entscheidungsfindung in der Entstehung der REST API.

So wurden Vor- und Nachteile verschiedener Komponenten in Bezug auf den vorliegenden Anwendungsfall aufgezeigt und abgewogen. Dabei wurde deutlich, dass in der Vielzahl an verschiedenen Programmiersprachen, Frameworks, API-Gestaltungsmöglichkeiten, Proxy-Servern et cetera darauf geachtet werden muss, wofür genau diese eingesetzt werden. Denn je nach Verwendungszweck können verschiedenen Komponenten unterschiedliche Qualitäten zum Gesamtsystem beitragen.

Im Hauptteil der Arbeit wurde eine zweckgerichtete REST API entworfen. Die benötigten Softwarekomponenten, die einen stabilen, schnellen und skalierbaren Umgang mit der API ermöglichen, wurden konfiguriert und es entstand ein schlüssiges Gesamtsystem. Dieses konnte durch die Verwendung von Docker zu einem einfach zu implementierenden Softwarepaket zusammengefasst werden, dass auf einer Vielzahl von Grundsystemen verwendet werden kann.

Die entstandene REST API ist nicht nur in der Lage, ohne große Konfiguration, Daten in einer MySQL-Datenbank zu verwalten, sie bietet auch Möglichkeiten zur Nutzerauthentifikation, Zugriffssicherung sowie Rechte- und Rollenverwaltung, um ein sicheres Datenmanagement zu gewährleisten. Diese Funktionen können in einem lokalen Netzwerk, aber auch durch Portweiterleitung über das Internet genutzt werden.

Im Konzeptteil wurde die API auf einem Einplatinencomputer (Raspberry Pi) installiert. Dieser wurde mit einem Temperatursensor ausgestattet, der als Beispiel für einen beliebigen Sensor stehen kann. So konnte einerseits die einfache Implementierung des Systems demonstriert werden, anderseits wurde gezeigt, wie eine Anbindung weiterer Python-Programme (zum Beispiel Sensorsteuerungsprogramme) möglich ist. Das Konzept ist schlussendlich ein Beispiel dafür, wie die entworfene API in der Praxis als schnelle günstige Lösung verwendet werden kann, um Daten zu erfassen und diese anschließend auf sinnvolle Weise in einem Netzwerk zugänglich zu machen beziehungsweise zu verwalten.

Es wird so eine Lösung für jeden geboten, der auf einfache Weise Maschinendaten durch Sensoren erheben, in einem Netzwerk zugänglich machen und den Zugriff zu diesen Daten durch verschiedene Personen oder Gruppen, regeln möchte.

Abbildungsverzeichnis

Tabellenverzeichnis

Code-Verzeichnis

Literaturverzeichnis

A Coding Project. (18. Mai 2017). *a-coding-project.de*. Von https://www.a-coding-project.de/ratgeber/http/request-methoden abgerufen

Aditya Mukerjee, A. V. (05. 05 2017). *CODE WORDS*. Von https://codewords.recurse.com/issues/five/what-restful-actually-means abgerufen

CA Security. (18. Mai 2017). *casecurity.org*. Von https://casecurity.org/ssl-basics/ abgerufen

Dan, J. (19. Mai 2017). *Microsoft Social Technet*. Von https://social.technet.microsoft.com/wiki/contents/articles/15189.difference-between-self-signed-ssl-certificate-authority.aspx abgerufen

DigiCert. (18. Mai 2017). *digicert.com*. Von https://www.digicert.com/ssl-cryptography.htm abgerufen

Django. (12. Juni 2017). *djangoproject.com*. Von https://docs.djangoproject.com/en/1.7/faq/general/#why-does-this-project-exist abgerufen

Docker. (22. Mai 2017). *docker.com*. Von https://www.docker.com/what-docker abgerufen

Docker. (22. Mai 2017). *docker.com*. Von https://www.docker.com/what-container abgerufen

Dwyer, G. (12. Juni 2017). *codementor.io*. Von https://www.codementor.io/garethdwyer/flask-vs-django-why-flask-might-be-better-4xs7mdf8v abgerufen

Eby, P. (20. Mai 2017). *Python.org*. Von https://www.python.org/dev/peps/pep-3333/#specification-overview abgerufen

Ellingwood, J. (06. Juni 2017). *digitalocean.com*. Von https://www.digitalocean.com/community/tutorials/apache-vs-nginx-practical-considerations abgerufen

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. Kalifornien.

Gourley, D., Totty, B., Sayer, M., Aggarwal, A., & Reddy, S. (2002). *HTTP: The Definitive Guide*. O'Reilly.

Griffiths, K. (14. Juni 2017). *http://blog.kgriffs.com*. Von http://blog.kgriffs.com/2012/12/18/uwsgi-vs-gunicorn-vs-node-benchmarks.html abgerufen

Grinberg, M. (2014). *Flask Web Development*. O'Reilly.

Hourieh, A. (2009). *Django 1.0*. Packt Publishing.

Iarocci, N. (20. Mai 2017). *Python-Eve*. Von http://python-eve.org/index.html abgerufen

James Snell, D. T. (2002). *Webservice-Programmierung mit SOAP*. O'Reilly .

json. (06. Juni 2017). *json.org*. Von http://www.json.org/json-de.html abgerufen

Lauterschlag, E. (16. Mai 2017). *web schmoekerer*. Von http://www.webschmoeker.de/grundlagen/http-hypertext-transfer-protocol/ abgerufen

MagPi. (04. April 2017). *raspberrypi.org*. Von https://www.raspberrypi.org/magpi/raspberry-pi-sales/ abgerufen

Microsoft. (19. Mai 2017). *Microsoft Support*. Von https://support.microsoft.com/de-at/help/257591/description-of-the-secure-sockets-layer-ssl-handshake abgerufen

Mikalauskas, A. (08. Juni 2017). *speedemy.com*. Von http://www.speedemy.com/apache-vs-nginx-2015/ abgerufen

Netcraft. (15. Mai 2017). *news.netcraft.com*. Von https://news.netcraft.com/archives/2017/04/21/april-2017-web-server-survey.html abgerufen

NGINX. (15. Mai 2017). *nginx.com*. Von https://www.nginx.com/resources/glossary/reverse-proxy-server/ abgerufen

Puglisi, S. (2015). *RESTful Rails Development: Building Open Applications and Services*. O'Reilly Media.

Raspberry Pi. (14. Juni 2017). *raspberrypi.org*. Von https://www.raspberrypi.org/help/faqs/#softwareLanguages abgerufen

Ruby on Rails. (12. Juni 2017). *rubyonrails.org*. Von http://guides.rubyonrails.org/getting_started.html#what-is-rails-questionmark abgerufen

Smith, B. (2015). *Beginning JSON*. Apress.

Soni, R. (2016). *Nginx: From Beginner to Pro*. Apress: Berkeley, CA.

Spring. (13. Juni 2017). *spring.io*. Von https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/html/overview.html abgerufen

SSL Shopper. (18. Mai 2017). *sslshopper.com*. Von https://www.sslshopper.com/what-is-ssl.html abgerufen

SSL Shopper. (18. Mai 2017). *sslshopper.com*. Von https://www.sslshopper.com/article-ssl-for-newbs.html abgerufen

Stringfellow, A. (05. Juni 2017). *stackify.com*. Von https://stackify.com/soap-vs-rest/ abgerufen

W3C. (30. Mai 2017). *w3.org*. Von https://www.w3.org/TR/2003/REC-soap12-part1-20030624/#intro abgerufen

W3C. (30. Mai 2017). *w3.org*. Von https://www.w3.org/TR/ws-arch/#id2263315 abgerufen

W3C. (2. Juni 2017). *w3.org*. Von https://www.w3.org/TR/REC-xml-names abgerufen

w3schools. (02. Juni 2017). *w3schools.com*. Von https://www.w3schools.com/xml/xml_whatis.asp abgerufen

w3schools. (2. Juni 2017). *w3schools.com*. Von https://www.w3schools.com/xml/xml_soap.asp abgerufen

Zwattendorfer, B. (2013). *Analyse SOAP vs. REST*. Graz: E-Government Innovationszentrum.