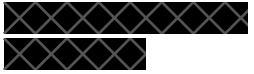


Effiziente Entwicklung eines RESTful Webservices auf Basis moderner Frameworks

B a c h e l o r a r b e i t

**Hochschule Hof
Fakultät Informatik
Studiengang Medieninformatik**

**Vorgelegt bei
Prof. Dr. Sven Rill
Alfons-Goppel-Platz 1
95028 Hof**

**Vorgelegt von
Steven Solleeder Isabell Waas**


Hof, 10.03.2023

Inhaltsverzeichnis

1	Ziel der Arbeit	1
2	Theoretischer Hintergrund	4
2.1	Übertragung von Webdaten	4
2.1.1	Webprotokoll HTTP	4
2.1.2	Webformat JSON	8
2.2	REST-Prinzip	11
2.3	Object Relational Mapping	12
2.4	Clienten zum Abrufen	13
3	Festlegung der Testbedingungen	16
3.1	Auswahl der Programmiersprachen und Frameworks	16
3.2	Aufgaben	20
3.2.1	Definition von Ressourcen und Data Transfer Objects	21
3.2.2	Abruf eines Objekts	23
3.2.3	Abruf aller Objekte mit Pagination, Sortierung und Filterung	25
3.2.4	Erstellung eines Objekts	27
3.2.5	Änderung einzelner Attribute eines Objekts	29
3.2.6	Lösung eines Objekts	31
3.2.7	Validierung von erhaltenen Daten	33
3.2.8	Behandlung von Fehlern	33
3.2.9	Konfiguration des Servers	34
3.3	Bewertungskriterien	35
3.3.1	Dokumentation und Wartung	35
3.3.2	Lines of Code	36
3.3.3	Notwendigkeit zusätzlicher Bibliotheken	38
3.3.4	Wichtige Architekturprinzipien und besondere Sprachfunktionen	39
3.3.5	Performance	40

4	Betrachtung und Bewertung der einzelnen Programmiersprachen und Frameworks	45
4.1	C# mit ASP.NET Core	46
4.1.1	Einführung in die Sprache und das Framework	46
4.1.2	Implementierung des RESTful Webservices	47
4.1.3	Bewertung	59
4.2	Java mit Spring Boot	68
4.2.1	Einführung in die Sprache und das Framework	68
4.2.2	Implementierung des RESTful Webservices	69
4.2.3	Bewertung	80
4.3	TypeScript mit Express.js	89
4.3.1	Einführung in die Sprache und das Framework	89
4.3.2	Implementierung des RESTful Webservices	90
4.3.3	Bewertung	105
4.4	Go mit Gin	114
4.4.1	Einführung in die Sprache und das Framework	114
4.4.2	Implementierung des RESTful Webservices	115
4.4.3	Bewertung	132
5	Auswertung der Erkenntnisse	140
5.1	Gegenüberstellung der Ergebnisse	140
5.2	Gesamtfazit	143

Abbildungsverzeichnis

1 Zeitlicher Verlauf des Suchinteresses an REST und SOAP weltweit laut Google Trends [15]	2
2 Das ISO/OSI-Schichtenmodell	5
3 Zeitlicher Verlauf des Suchinteresses an XML und JSON weltweit laut Google Trends [14]	9
4 Bildschirmfoto der Desktop-Anwendung von Postman	13
5 Erste 20 Plätze des PYPL-Index [3]	17
6 Frameworks für C# [8]	19
7 Frameworks für Java [8]	19
8 Frameworks für JavaScript [8]	19
9 Frameworks für Go [8]	19
10 Spezifikation der Schemata	21
11 Spezifikation des Endpunkts zum Abrufen eines Buches	23
12 Spezifikation des Endpunkts zum Abrufen aller Bücher	25
13 Spezifikation des Endpunkts zum Erstellen eines Buches	27
14 Spezifikation des Endpunkts zum Aktualisieren eines Buches	29
15 Spezifikation des Endpunkts zum Löschen eines Buches	31
16 Übersicht der Spezifikation	32
17 C# Logo (inoffiziell) [37]	46
18 .NET Core Logo (inoffiziell) [32]	46
19 Seite zu einem der größeren Entwicklungsthemen ASP.NET Core von Microsoft [19]	59
20 Ein Artikel der Artikelsammlung von ASP.NET Core [5]	60
21 Releases von .NET [17]	62
22 Architektur des Webservices mit ASP.NET Core	64
23 Ausgewählte Antwortzeiten für C# mit ASP.NET Core (Balkenwerte manuell hinzugefügt)	66
24 Ausgewählte Sessionlängen für C# mit ASP.NET Core (Balkenwerte manuell hinzugefügt)	67
25 Java Logo [34]	68
26 Spring Logo [35]	68

27 Übersichtsseite der Module von Spring [11]	80
28 Reference Documentation von Spring Boot [29]	81
29 Api-Dokumentation von Spring Boot 3.0.3 [12]	82
30 Releases von Spring Boot [11]	83
31 Architektur des Webservices mit Spring Boot	85
32 Ausgewählte Antwortzeiten für Java mit Spring Boot (Balkenwerte manuell hinzugefügt)	88
33 Ausgewählte Sessionlängen für Java mit Spring Boot (Balkenwerte manuell hinzugefügt)	88
34 TypeScript Logo [38]	89
35 Express.js Logo [23]	89
36 Guide zum Thema Routing in Express [23]	105
37 API Reference von Express 4.x [23]	106
38 Releases von Express [27]	107
39 Architektur des Webservices mit Express	109
40 Ausgewählte Antwortzeiten für TypeScript mit Express (Balkenwerte manuell hinzugefügt)	112
41 Ausgewählte Sessionlängen für TypeScript mit Express (Balkenwerte manuell hinzugefügt)	113
42 Go Logo [33]	114
43 Gin Logo [25]	114
44 Ausschnitt aus dem Gin Quick Start Guide	133
45 Ausschnitt aus der Packagebeschreibung von Gin	134
46 Architektur des Webservices mit Gin	136
47 Ausgewählte Antwortzeiten für Go mit Gin (Balkenwerte manuell hinzugefügt)	139
48 Ausgewählte Sessionlängen für Go mit Gin (Balkenwerte manuell hinzugefügt)	139
49 Vergleich der Programmiersprachen und Frameworks anhand der fünf Bewertungskriterien	141

Listingsverzeichnis

1 Beispiel eines JSON-Dokuments	10
2 Beispiel für ein curl-Kommando samt Rückgabe in einer Shell	14
3 Codebeispiel zur Veranschaulichung der SLOC und LLOC Metriken	37
4 Testskript zur Messung der Leistung	42
5 Kommando zum Durchführen des Artillery-Testskripts	44
6 Kommando zum Umwandeln eines Artillery-Ergebnisses in eine HTML-Datei	44
7 C#-Code des DTOs CreateBook	47
8 C#-Code der Model-Klasse Book	48
9 C#-Code für den BooksContext	48
10 C#-Code der Registrierung des DbContexts	49
11 C#-Code des BooksControllers mit der Methode GetBook	50
12 C#-Code der Controller-Methode GetBooks	51
13 C#-Code der Controller-Methode CreateBook	52
14 C#-Code der Controller-Methode UpdateBook	53
15 C#-Code der Controller-Methode DeleteBook	54
16 C#-Code eines Attributs zur Validierung einer ISBN-Nummer	55
17 C#-Code eines ExceptionHandlers	56
18 Datei launchSettings.json des .NET-Projekts	56
19 Datei appsettings.json des .NET-Projekts	57
20 C#-Code der Datei Program.cs	58
21 Nutzen der Methode AddDbContext in der Datei Program.cs	63
22 Java-Code der Ressource Book	69
23 Java-Code des BooksRepositorys	71
24 Java-Code des BooksController mit der Methode getBook	72
25 Java-Code der Controller-Methode getAllBooks	73
26 Java-Code der Controller-Methode createBook	74
27 Java-Code der Controller-Methode updateBook	75
28 Java-Code der Controller-Methode deleteBook	75
29 Java-Code des ErrorHandlers	77
30 Groovy-Code der Datei build.gradle	78

31 Groovy-Code der Datei settings.gradle	78
32 Datei application.properties des Spring Boot-Projekts	79
33 Java-Code der Datei LibraryApiApplication.java	79
34 TypeScript-Code des Typs Book	90
35 TypeScript-Code des Typs CreateBook	90
36 TypeScript-Code des Interfaces DataAccessor	91
37 TypeScript-Code des BooksService mit der Methode readById	92
38 TypeScript-Code des BooksControllers mit der Methode getBook	93
39 TypeScript-Code des BooksRouters	94
40 TypeScript-Code der Service-Methode read	96
41 TypeScript-Code der Controller-Methode getAllBooks	96
42 TypeScript-Code der Service-Methode create	97
43 TypeScript-Code der Controller-Methode createBook	97
44 TypeScript-Code der Service-Methode update	98
45 TypeScript-Code der Controller-Methode updateBook	99
46 TypeScript-Code der Service-Methode delete	99
47 TypeScript-Code der Controller-Methode deleteBook	99
48 TypeScript-Code der FormValidator-Methode isUpdateBook	100
49 TypeScript-Code zweier ContentValidator-Methoden	101
50 TypeScript-Code des ErrorHandlers	101
51 Datei package.json für das Node-Projekt	103
52 TypeScript-Code der Datei index.ts	103
53 Go-Code des DTOs CreateBook	116
54 Go-Code des Interfaces DataAccessor	116
55 Go-Code des BooksServices mit der Methode ReadById	117
56 Go-Code der Controller-Funktion GetBook	118
57 Go-Code der Funktion SetupRouter	119
58 Go-Code der Service-Methode Read	120
59 Go-Code der Controller-Funktion GetBooks	121
60 Go-Code der Service-Methode Create	122
61 Go-Code der Controller-Funktion CreateBook	122

VIII

62 Go-Code der Service-Methode Update	123
63 Go-Code der Controller-Funktion UpdateBook	125
64 Go-Code der Service-Methode Delete	125
65 Go-Code der Controller-Funktion DeleteBook	126
66 Go-Code zur Validierung der Id in den Endpunkten	127
67 Go-Code der Methoden BindQuery und BindJson und bind	128
68 Go-Code zur Validierung in einer Controller Funktion	129
69 Go-Code eines Error-Aufrufs	129
70 Go-Code der Error Middleware	130
71 Go-Code der main-Funktion	130

1 Ziel der Arbeit

Steven Solleder

In der heutigen Zeit sind Webanwendungen aus unserem Leben nicht mehr wegzudenken. Sowohl im beruflichen als auch im privaten Umfeld sind Applikationen wie Onlineshops, Kommunikationsplattformen und mehr im Einsatz. Viele der Webseiten oder Anwendungen, die täglich von zahlreichen Menschen genutzt werden, greifen im Hintergrund auf Informationen zu, die von weiteren Diensten über das Internet zur Verfügung gestellt werden. So kann es beispielsweise sein, dass eine Webanwendung zum Planen einer Urlaubsreise die Sehenswürdigkeiten, Hotels, Flug- oder Wetterdaten, die sie dem Anwender bereitstellt, alle von unterschiedlichen Schnittstellen im Web abruft. Für diese Kommunikation, die der Nutzer in der Regel nicht mitbekommt, gibt es Webservices.

Ein Webservice dient im Allgemeinen dazu, eine Schnittstelle (API) im Web zu definieren. Diese kann dann wiederum durch einen Client, das heißt in der Regel einen Browser, einen Webserver oder einen anderen Webservice verwendet werden. Nachdem eine Anfrage an die API gestellt wurde, führt diese eine Aktion aus und sendet anschließend eine Antwort zurück [1]. Durch die Nutzung eines Webservices ergeben sich gleich zwei Vorteile: Zum einen können Webservices plattformunabhängig eingesetzt werden. Das bedeutet insbesondere, dass die beiden miteinander kommunizierenden Anwendungen nicht mit denselben Programmiersprachen und Frameworks entwickelt sein müssen. Zum anderen kapselt ein Webservice eine bestimmte Menge an Daten und Funktionalität und stellt diese für beliebig viele Clienten über das Internet bereit. Damit kann unter anderem viel Zeit und Programmieraufwand gespart werden, da die Clienten die in die Schnittstelle ausgelagerte Funktionalität wiederverwenden können und nicht neu implementieren müssen [46].

Bei der Umsetzung von Webservices wird auf unterschiedliche Konzepte beziehungsweise Protokolle zurückgegriffen. Die bekanntesten und schon seit Jahren am häufigsten genutzten sind SOAP und Representational State Transfer (REST). Bei SOAP handelt es sich um ein Netzwerkprotokoll, dessen Name ursprünglich ein Akronym für „Simple Object Access Protocol“ war, jedoch mittlerweile nur noch für sich selbst steht, da die Bezeichnung sich als unpassend herausstellte. Es liegt, insbesondere was Sicherheit und Fehlerbehand-

lung angeht, vor seinem etwas neueren Konkurrenten. Der Architekturstil REST, der noch in einem eigenen Kapitel dieser Arbeit genauer behandelt wird, ist hingegen jedoch wesentlich leichtgewichtiger, dadurch auch schneller zu entwickeln und arbeitet im Gegensatz zu SOAP nicht nur mit XML, sondern auch mit anderen Datenformaten. Aus diesen Gründen wird REST heutzutage bevorzugt, während SOAP primär in Altsystemen genutzt wird [45]. Dass REST über die Jahre auf der ganzen Welt deutlich an Beliebtheit gewonnen hat und SOAP dagegen immer weniger von Interesse ist, zeigt auch das nachfolgende Bildschirmfoto von Google Trends, in welchem REST und SOAP anhand der Anzahl ihrer Suchanfragen verglichen werden.



Abbildung 1: Zeitlicher Verlauf des Suchinteresses an REST und SOAP weltweit laut Google Trends [15]

Mittlerweile existieren unzählige REST APIs von teilweise auch sehr bekannten Unternehmen. So gibt es zum Beispiel Webservices des Marktplatzes eBay¹, des Kartendienstes Bing Maps von Microsoft² sowie des Bezahldienstes PayPal³. Viele dieser Services stehen Nutzern sogar kostenlos zur Verfügung oder erlauben zumindest eine bestimmte Menge an Anfragen pro Monat, bevor für weitere Geld verlangt wird.

¹<https://developer.ebay.com/develop/apis/restful-apis>

²<https://learn.microsoft.com/en-us/bingmaps/rest-services/>

³<https://developer.paypal.com/api/rest/>

Webservices, welche das REST-Prinzip umsetzen, werden folglich immer häufiger benötigt. Dabei müssen bestimmte Anforderungen bei der Programmierung immer umgesetzt werden. Hierzu zählen zum Beispiel Routing und die Validierung von Daten. Soll nun ein sogenannter RESTful Webservice entwickelt werden, so stehen zahlreiche Programmiersprachen und für diese wiederum verschiedenste Frameworks zur Auswahl. Letztere bringen bereits viele oft benötigte Features mit sich und erleichtern so unter anderem das Parsen von Daten, das Routen von Uniform Resource Locators (URL) oder das Sorgen für Sicherheit und eine gute Fehlerbehandlung. Neben bereits seit vielen Jahren etablierten, sehr großen Frameworks wie Spring⁴ für Java⁵ gibt es dabei auch noch ziemlich junge Konkurrenten. Hier ist beispielsweise Ktor⁶ für die Programmiersprache Kotlin⁷ zu nennen. All diese Frameworks und selbstverständlich auch die gemeinsam mit ihnen verwendeten Programmiersprachen unterscheiden sich in vielerlei Hinsicht stark.

Aus diesen Gründen ist das Ziel dieser Arbeit zu ermitteln, welche Programmiersprache und welches Framework sich am besten eignen, um einen RESTful Webservice umzusetzen. Hierzu werden verschiedenste, bei derartigen Services typischerweise benötigte Anforderungen nacheinander in unterschiedlichen Frameworks umgesetzt. Anhand einiger in einem späteren Kapitel beschriebener Kriterien wird dann untersucht, wie gut die Entwicklung mit dem aktuell beleuchteten Framework und der dabei verwendeten Programmiersprache funktioniert. Schließlich sollen die Ergebnisse verglichen werden und es soll herausgefunden werden, welche Kombination aus Programmiersprache und Framework am besten geeignet ist.

⁴<https://spring.io/>

⁵<https://dev.java/>

⁶<https://ktor.io/>

⁷<https://kotlinlang.org/>

2 Theoretischer Hintergrund

Isabell Waas

Bevor sich dem eigentlichen Inhalt der Arbeit gewidmet werden kann, müssen zunächst ein paar wichtige Begriffe und Technologien erläutert werden, die im weiteren Verlauf vorausgesetzt werden.

2.1 Übertragung von Webdaten

2.1.1 Webprotokoll HTTP

Isabell Waas

In einem Rechnernetz, welches eine Vereinigung mehrerer eigentlich unabhängiger Systeme wie Computern oder Druckern darstellt, sind zur Kommunikation zwischen den verschiedenen Geräten Protokolle notwendig. Diese legen die valide Syntax, das heißt, die Form, und Semantik, also den Inhalt, von bei der Kommunikation eingesetzten Nachrichten fest. Da die Kommunikation aufgrund der großen Anzahl an Anforderungen, beispielsweise an die Sicherheit oder Zuverlässigkeit, sehr komplex ist, wurden Schichtenmodelle eingeführt, die den Prozess abstrakt beschreiben. Eines der verbreitetsten Modelle ist das 1983 standardisierte ISO/OSI-Modell, das in der Grafik auf der folgenden Seite zu sehen ist [2].

Wie Abbildung 2 zeigt, besteht das ISO/OSI-Modell aus sieben Schichten. Jede Schicht setzt dabei bestimmte Anforderungen an die Kommunikation um und beinhaltet verschiedene Protokolle. Ebenso nutzt sie jeweils Dienste der Schicht unter ihr und stellt wiederum eine Schnittstelle für die unmittelbar darüberliegende Schicht bereit.

- **Bitübertragungsschicht:** Die unterste Schicht beschäftigt sich mit der physikalischen Datenübertragung über Signale, welche beispielsweise elektrisch oder optisch sein können [2].
- **Sicherungsschicht:** Die zweite Schicht sorgt für eine zuverlässige Verbindung auf dem aktuellen Übertragungsabschnitt und regelt den Zugriff auf das Übertragungsmedium. Auch korrigiert sie Übertragungsfehler mithilfe von Prüfsummen.
- **Vermittlungsschicht:** In der Vermittlungsschicht, die manchmal auch Internetschicht genannt wird, stehen Adressierung und Routing im Vordergrund.

- **Transportschicht:** Die Transportschicht kümmert sich um eine sichere Ende-zu-Ende Verbindung zwischen zwei Anwendungsprozessen und achtet auf eine richtige Reihenfolge bei der Datenübertragung.
- **Sitzungsschicht:** Die Aufgabe dieser Schicht ist das Auf- und Abbauen einer virtuellen Verbindung zwischen zwei Anwendungsprozessen, eine sogenannte Sitzung.
- **Darstellungsschicht:** Die Darstellungsschicht sorgt für eine standardisierte Informationsabbildung. Dazu werden zum Beispiel Daten komprimiert oder verschlüsselt [2].
- **Anwendungsschicht:** In der obersten Schicht werden anwendungsspezifische Kommunikationsdienste zur Verfügung gestellt. Dazu gehören zum Beispiel Dienste zur Datenübertragung oder zur Synchronisierung. Zudem findet hier die Ein- und Ausgabe der eigentlichen Nachricht, z. B. einer E-Mail, statt [2].

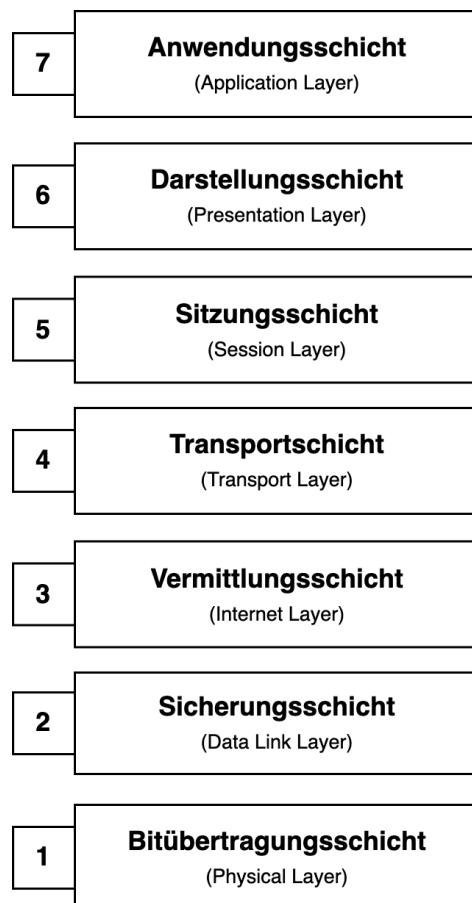


Abbildung 2: Das ISO/OSI-Schichtenmodell

Im Folgenden soll die Anwendungsschicht genauer beleuchtet werden. Grund dafür ist, dass es in der vorliegenden Arbeit um das REST-Paradigma (siehe Kapitel 2.2) gehen soll und dieses vor allem durch die in dieser Schicht verwendeten Protokolle umgesetzt wird. Auch wenn REST kein bestimmtes Protokoll auf der Anwendungsschicht verlangt, wird es nahezu ausschließlich in Verbindung mit dem Hypertext Transfer Protocol (HTTP) oder dem darauf aufbauenden Hypertext Transfer Protocol Secure (HTTPS) verwendet [40]. Daher soll im Folgenden näher auf HTTP eingegangen werden.

Der häufigste Anwendungsfall des HTTP Protokolls ist die Kommunikation eines Browsers auf Clientseite mit einem Server beim Laden einer Webseite im World Wide Web (WWW). Hierbei finden zahlreiche Anfragen statt, da bei einer klassischen Webseite jede eingebundene Datei, das heißt zum Beispiel ein Bild oder JavaScript-Code, einzeln abgerufen werden muss. Daneben wird das Protokoll aber auch noch in anderen Szenarien eingesetzt, zum Beispiel im Rahmen eines RESTful Webservices [1].

Im Allgemeinen dient HTTP dazu, Daten zu übertragen. Es definiert hierfür, wie die Kommunikation zwischen einem Client und Server aussieht. Die Kommunikation wird stets durch den Client begonnen, indem er eine HTTP-Anfrage, die auch Request genannt wird, an den Server stellt. Erhält der Server eine Anfrage, so bearbeitet er sie in der Regel durch das Ausführen von Logik wie z. B. einer Datenbankoperation. Anschließend übermittelt er dem Client eine Antwort, die sogenannte Response. Wie ein Request- beziehungsweise ein Response-Objekt aussieht, ist ebenfalls durch das Protokoll vorgeschrieben. Zusammengefasst kann gesagt werden, dass beide Objekte jeweils einen Nachrichtenkopf, die Header, sowie einen Nachrichtenkörper, den Body, besitzen und sich in ihrem Aufbau sehr ähnlich sind. Während in den Headern Meta-Informationen wie beispielsweise welche Zeichensätze und Sprachen der Client akzeptiert angegeben werden, befinden sich im Body die tatsächlichen Daten, das heißt zum Beispiel ein Hypertext Markup Language (HTML)-Dokument oder Daten im Format JavaScript Object Notation (JSON), das im nächsten Kapitel erläutert wird [2, 1]. HTTP-Antworten beinhalten zudem einen Statuscode, eine dreistellige Zahl, die einer von fünf definierten Kategorien angehört, welche jeweils mit den Ziffern 1 bis 5 beginnen. So stehen z. B. die mit 2 startenden Statuscodes für einen Erfolg und diejenigen, deren erste Ziffer 4 ist, für einen Fehler auf Clientseite. Jeder Statuscode besitzt einen kurzen Beschreibungstext. Einer der bekanntesten Statuscodes ist 404 mit der Beschreibung "Not Found" [1].

Zur Übertragung der Daten nutzt HTTP ein Transportprotokoll. Dabei wird meistens das Transmission Control Protocol (TCP) eingesetzt. Dieses ist im Gegensatz zu seiner verbreitetsten Alternative, dem User Datagram Protocol (UDP), zuverlässiger und stellt sicher, dass alle Informationen beim Empfänger ankommen [2].

Als Nächstes soll auf die Methoden eingegangen werden, welche meist HTTP-Verben genannt werden und in Bezug auf REST-APIs eine große Rolle spielen (siehe Kapitel 2.2). Jede Anfrage muss ein Verb beinhalten, damit der Server weiß, wie er auf sie reagieren soll. Die wichtigsten fünf HTTP-Verben sind GET zum Abrufen von Daten, POST zum Senden von Daten, PUT zum Aktualisieren eines gesamten Datensatzes, PATCH zum Update von Teilen eines Datensatzes und DELETE zum Löschen von Daten.

Eine der Eigenschaften von HTTP, die darüber hinaus stets betont wird, ist seine Zustandslosigkeit. Das bedeutet, dass jede HTTP-Anfrage unabhängig von den vorherigen und nachfolgenden Anfragen betrachtet wird und der Server somit nicht wissen kann, ob er mehrfach mit einem bestimmten Client oder mit vielen verschiedenen Clienten kommuniziert. Ist dies jedoch relevant, beispielsweise um den Nutzer in seinem Account eingeloggt zu lassen, auch wenn zwischenzeitlich das Browserfenster geschlossen wird, so können Cookies genutzt werden. Mit diesen können ausgewählte Daten auf Clientseite im Browserspeicher für eine bestimmte Zeitdauer gesichert werden. So kann also ein Authentifizierungstoken im Cookie gesichert und bei jedem Request mitgesendet werden, sodass der Server den Nutzer wiedererkennt [1].

Zuletzt soll noch angesprochen werden, dass auf Basis des HTTP-Protokolls auch bidirektionale Kommunikation realisiert werden kann, obwohl es grundsätzlich für unidirektionale Kommunikation vorgesehen ist. Auch wenn es noch ein paar andere Möglichkeiten gibt, Daten mehr oder weniger aktiv vom Server an den Client zu senden, so ist die Nutzung des WebSocket-Protokolls die einzige, welche tatsächlich bidirektional funktioniert [1]. Da jedoch im Rahmen dieser Arbeit bidirektionale Kommunikation keine Rolle spielt, soll diese nicht näher erläutert werden.

2.1.2 Webformat JSON

Isabell Waas

Abgesehen von HTML, Cascading Style Sheets (CSS) und JavaScript, die typischerweise für Webseiten benötigt werden, werden über HTTP auch Daten in weiteren Formaten ausgetauscht. Hierbei wird zwischen Bildformaten wie zum Beispiel Joint Photographic Experts Group (JPEG) oder Portable Network Graphics (PNG), Multimediaformaten wie MP4 und Datenformaten unterschieden [1]. Auf Letzteren soll in diesem Kapitel der Fokus liegen, da sie insbesondere bei Webservices intensiv verwendet werden. Vorher soll jedoch noch der Begriff Multipurpose Internet Mail Extension Type (MIME-Type) eingeführt werden, der bei der Verwendung verschiedener Formate eine große Bedeutung hat.

Der MIME-Type ist ein kurzer Text, welcher das Format einer Datei definiert. Er besteht aus einem Typ, zum Beispiel „text“ oder „image“, einem Schrägstrich und einem Subtyp, welcher das konkrete Format angibt. Ein korrekter MIME-Type wäre somit „text/html“ oder „image/png“. Bei der Kommunikation über HTTP kommt der MIME-Type zum Einsatz, um dem Client beziehungsweise dem Server mitzuteilen, welches Format die Daten im Nachrichtenbody besitzen. Es gibt dafür einen Header namens „Content-Type“, der in einem Request- oder Response-Objekt mitgeschickt werden kann [1].

Sollen komplexe Daten, insbesondere welche mit mehreren Verschachtelungen, über das Web übermittelt werden, so werden hauptsächlich die zwei bekannten Datenformate Extensible Markup Language (XML) und JSON verwendet. Grundsätzlich ähneln sich die beiden Formate, da sie hierarchisch aufgebaut und gut lesbar sind. Zudem können beide geparsst und so gemeinsam mit vielen Programmiersprachen genutzt werden. Dennoch ist JSON seit einigen Jahren beliebter als XML, was hauptsächlich an seiner kürzeren und einfacheren Syntax liegt und der Tatsache, dass es mit weniger Aufwand in JavaScript genutzt werden kann. Die folgende Grafik zeigt ein Bildschirmfoto von Google Trends, auf dem die Entwicklung der Anzahl der Suchanfragen der Begriffe „json“ und „xml“ über die letzten 16 Jahre weltweit verglichen wird [1, 6].

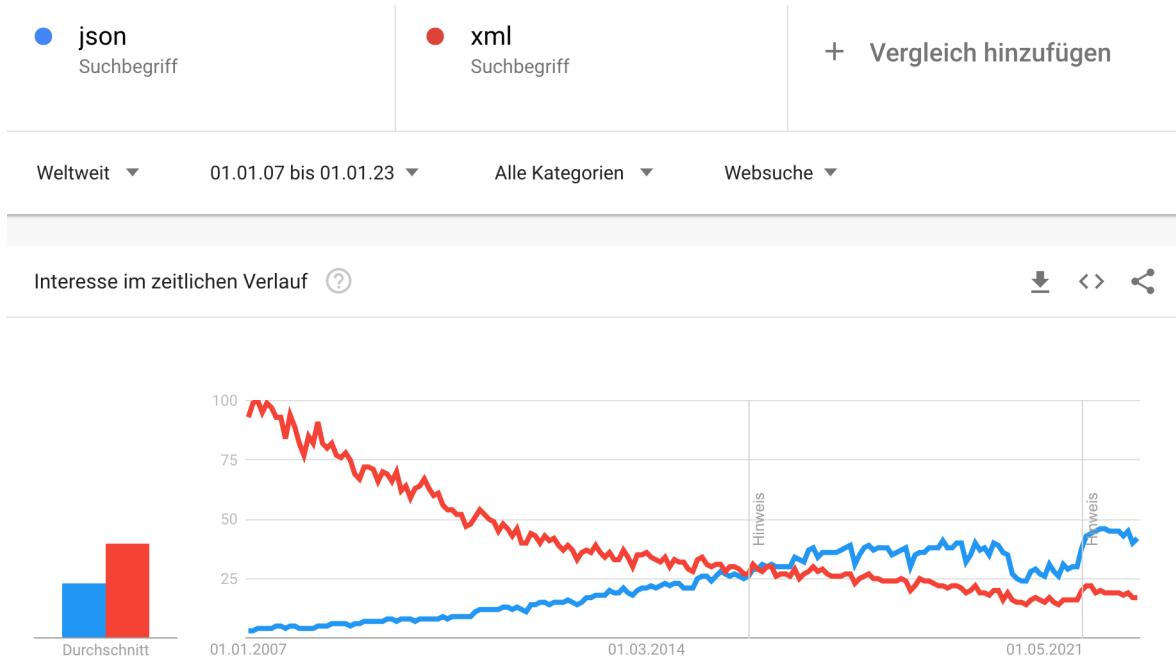


Abbildung 3: Zeitlicher Verlauf des Suchinteresses an XML und JSON weltweit laut Google Trends [14]

Wie in der Abbildung zu sehen ist, ist die Beliebtheit von JSON kontinuierlich gestiegen und liegt seit einiger Zeit stetig über dem Interesse an XML. Aus diesem Grund wird auch in den RESTful Services im Rahmen dieser Arbeit JSON als Datenformat verwendet.

Charakteristisch für JSON sind die geschweiften Klammern, welche einzelne Objekte von einander abgrenzen. Innerhalb dieser werden verschiedene Eigenschaften eines Objektes in der Regel durch Schlüssel-Wert-Paare definiert. Jeder Schlüssel sowie Werte, welche Texte darstellen, stehen dabei in Anführungszeichen. Zahlen und Wahrheitswerte stehen nicht in Anführungszeichen. Zudem kann ein Wert auch ein weiteres Objekt, welches wiederum innerhalb von geschweiften Klammern steht, oder ein Array sein. Letztere werden durch eckige Klammern definiert. Insgesamt ähnelt die Syntax in einem JSON-Dokument der Syntax von JavaScript sehr [1]. Im nachfolgenden Listing ist ein beispielhaftes JSON-Dokument zu sehen, welches die Syntax verdeutlichen soll.

```

1  {
2      "movies": [
3          {
4              "id": 1,
5                  "title": "Harry Potter and the Philosopher's Stone",
6                  "genres": [
7                      {
8                          "id": 2,
9                              "name": "Fantasy"
10                         },
11                         {
12                             "id": 3,
13                                 "name": "Adventure"
14                         }
15                     ],
16                     "year": 2001,
17                     "director": "Chris Columbus"
18                 },
19                 {
20                     "id": 5,
21                     "title": "Star Wars: Episode IV – A New Hope",
22                     "genres": [
23                         {
24                             "id": 1,
25                             "name": "Science Fiction"
26                         },
27                         {
28                             "id": 3,
29                             "name": "Adventure"
30                         }
31                     ],
32                     "year": 1978,
33                     "director": "George Lucas"
34                 }
35             ]
36 }

```

Listing 1: Beispiel eines JSON-Dokuments

2.2 REST-Prinzip

Steven Solleder

Als Nächstes soll nun das bereits mehrfach erwähnte REST-Prinzip näher erläutert werden, welches in dieser Arbeit eine zentrale Rolle spielt. Es wurde im Jahr 2000 von dem Informatiker Roy T. Fielding ins Leben gerufen. Das REST-Paradigma beschäftigt sich insbesondere mit der Architektur von verteilten Systemen, zu denen Webservices gehören. Hierbei bilden Ressourcen die wichtigsten Bestandteile, welche über sogenannte Endpunkte mithilfe des bereits ausführlich beschriebenen HTTP- beziehungsweise HTTPS-Protokolls abrufbar sein sollen [7, 40]. Dabei umfasst der REST-Architekturstil die folgenden fünf Prinzipien:

1. **Eindeutige Addressierbarkeit jeder Ressource:** Auf jede Ressource muss über eine eindeutige Adresse, eine URL, zugegriffen werden können. Ein Beispiel für eine solche URL könnte „<http://www.messenger.de/tim-taler/profilbild>“ sein.
2. **Verschiedene Repräsentationen einer Ressource:** Im vorherigen Kapitel wurde bereits von verschiedenen Webformaten wie JSON und XML gesprochen. Das REST-Paradigma erlaubt, dass eine Ressource in mehreren Formaten vorliegt. Soll ein bestimmtes Format abgerufen werden, so kann dies über spezielle Header bei der Anfrage angegeben werden [4].
3. **Zustandslose Anfragen:** Die Kommunikation mit Servern, welche nach dem REST-Paradigma arbeiten, ist zustandslos, was unter anderem daran liegt, dass fast ausschließlich das HTTP-Protokoll eingesetzt wird. Wie in Kapitel 2.1.1 erläutert wurde, bedeutet Zustandslosigkeit, dass bei einer Anfrage keine Informationen mehr über vorherige Anfragen vorliegen und stets nur die aktuell im Request vorhandenen Daten genutzt werden können.
4. **Selbsterklärende Schnittstellen:** Ein RESTful Webservice nutzt stets bestimmte, standardisierte Bestandteile des HTTP-Protokolls. Dazu gehören die HTTP-Verben wie GET, POST und DELETE sowie definierte Statuscodes, darunter beispielsweise der Code 200, welcher eine erfolgreiche Ausführung der Anfrage anzeigt (siehe Kapitel 2.1.1). Durch die Verwendung etablierter Standards ist die Kommunikation mit dem Service leicht verständlich [7, 40].
5. **Verlinkungen durch Hypermedia:** Zuletzt sieht REST vor, dass durch Verlinkungen durch die verschiedenen Ressourcen navigiert werden kann. In den Antworten des

Servers sind daher je nach Datenformat der zurückgelieferten Ressource Attribute wie z. B. „href“ zu finden, welche Adressen anderer Ressourcen beinhalten [7, 40].

Genauere Informationen zu REST und vor allem zu dessen fünf Hauptprinzipien können in der Praxisarbeit nachgelesen werden [48].

2.3 Object Relational Mapping

Isabell Waas

Wenn eine Software Daten dauerhaft speichern soll, so verwendet sie meist eine Datenbank. Hierbei ist anzumerken, dass es zwei Typen von Datenbanken gibt: die relationalen und die nicht-relationalen Datenbanken. Erstere basieren auf dem am häufigsten genutzten Datenbankmodell, weshalb in der vorliegenden Arbeit das bekannte relationale Datenbankmanagementsystem MariaDB⁸ verwendet wird [41].

Bei relationalen Datenbanken stehen die als Relationen bezeichneten Tabellen im Fokus, die jeweils Daten eines bestimmten Entitätstypen speichern. Die Attribute des Entitätstypen werden dabei über die Spalten realisiert. Jeder Eintrag in einer Tabelle stellt eine Entität dar und muss eindeutig identifizierbar sein. Des Weiteren werden Beziehungen zwischen mehreren Entitäten entweder durch Attribute oder separate Tabellen realisiert.

Um Daten aus einer relationalen Datenbank abzurufen sowie zu verändern, wird die Abfragesprache Structured Query Language (SQL) eingesetzt. Dies ist jedoch insbesondere bei der Verwendung mit objektorientierten Programmiersprachen sehr umständlich, da die Daten stets händisch zwischen dem Format der Datenbank und dem spezifischen Objektformat der Sprache umgewandelt werden müssen. Vor allem bei verschachtelten Entitäten treten hierbei auch oft Fehler auf. Aus diesem Grund gibt es Verfahren zur automatisierten Konvertierung zwischen Relationen und Objekten, z. B. Object-Relational Mapping (ORM). Bei der Verwendung von ORM gibt es eine Abstraktionsschicht zwischen den Datensätzen und Objekten, die für die automatische Umwandlung der Daten sorgt. Dadurch müssen Datenbankabfragen auch nicht mehr in SQL verfasst werden, sondern können in der objektorientierten Programmiersprachen erfolgen. Insgesamt erleichtert ORM die Programmierung sehr, weshalb bei den Frameworks, die in dieser Arbeit betrachtet werden, insofern ein ORM eingebaut oder explizit empfohlen ist, dieses auch genutzt wird (siehe Kapitel 3.3.3).

⁸<https://mariadb.com/de/>

Nähere Informationen zu relationalen Datenbanken, insbesondere auch zu MariaDB, sowie zu ORM sind in der Praxisarbeit zu finden [48].

2.4 Clienten zum Abrufen

Steven Solleeder

Nun sollen mögliche Clienten betrachtet werden, mit denen HTTP-Endpunkte getestet werden können. Wie bereits beschrieben, ist HTTP ein standardisiertes Protokoll. Dadurch ist es prinzipiell jedem möglich, sowohl eigene Clienten als auch eigene Server zu schreiben, ganz unabhängig von der verwendeten Programmiersprache, dem Betriebssystem oder dem Framework. Dementsprechend gibt es zahlreiche Clienten zum Senden und Empfangen von HTTP-Anfragen. Im Folgenden sollen der populärste grafische und textbasierte Client näher beschrieben werden.

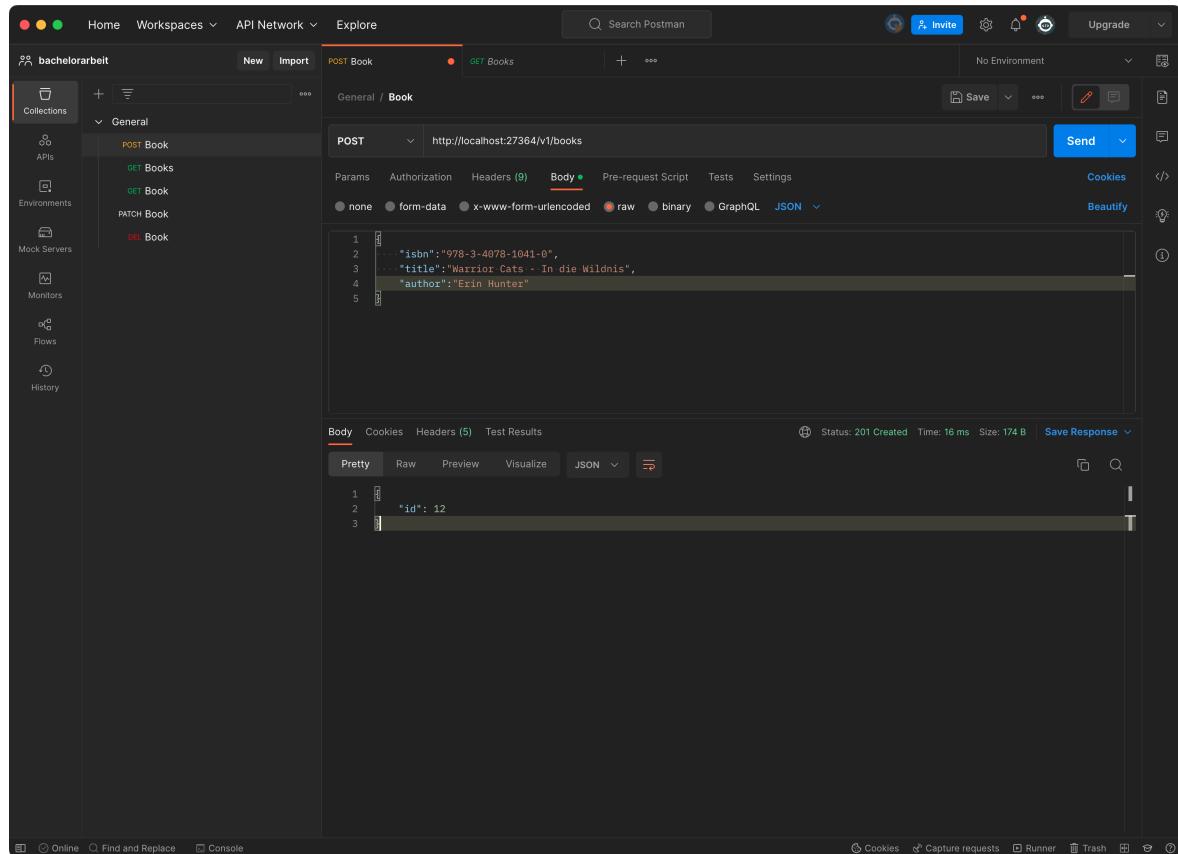


Abbildung 4: Bildschirmfoto der Desktop-Anwendung von Postman

Der wohl bekannteste grafische HTTP-Client ist Postman⁹, dessen Desktop-Anwendung auf der vorherigen Seite zu sehen ist. Postman wurde ursprünglich 2012 als Nebenprojekt von Abhinav Asthana entwickelt. Das Ziel der Anwendung war es, die API-Entwicklung zu vereinfachen [43].

Zum einen erlaubt Postman es, jeden Aspekt einer Anfrage, z. B. die Header oder den Body, individuell einzustellen. Zur Vereinfachung werden einige Header automatisch generiert. Ebenso gibt es für häufige Aspekte von HTTP-Anfragen eigene GUI-Elemente, wie beispielsweise für das Ändern einer Pfadvariable oder der Authentifizierung.

Zum anderen versucht Postman, Entwicklern das Arbeiten mit HTTP-Endpunkten so einfach wie möglich zu gestalten, indem es noch einige weitere Funktionen anbietet. Darunter befinden sich das Erstellen von Tests, das Schreiben von OpenAPI-Dokumenten oder das Nutzen und Austauschen von Umgebungsvariablen. Das alles und mehr ist auch in geteilten Workspaces möglich, sodass jeder Beteiligte stets auf dem neuesten Stand ist, was besonders bei größeren Teams sehr viel Zeit erspart.

Wer lieber in der Shell arbeitet oder auf einem Server ohne Desktopumgebung und Window-Manager hat natürlich auch andere Möglichkeiten. Ein Beispiel ist das äußerst populäre Programm curl¹⁰. Ein Beispiel für ein curl-Kommando innerhalb einer Shell ist unten dargestellt.

```

1 curl -X 'POST' \
2   -H 'Content-Type: application/json' \
3   -d '{
4     "isbn":"978-3-4078-1041-0",
5     "title ":"Warrior Cats – In die Wildnis",
6     "author ":"Erin Hunter"
7   }' \
8   -k \
9   "http://localhost:27364/v1/books"
10 {"id":13}%

```

Listing 2: Beispiel für ein curl-Kommando samt Rückgabe in einer Shell

⁹<https://www.postman.com/>

¹⁰<https://curl.se/>

curl beschränkt sich bei dem Erstellen von Anfragen auf das Wesentliche. Wie auch bei Postman ist es möglich, jeglichen Aspekt bezüglich einer HTTP-Anfrage festzulegen. So kann mit der Option -X das HTTP-Verb gesetzt, mit -H ein Header konfiguriert und mit der Option -d der Wert des Bodys festlegt werden. In dieser Arbeit geht es hauptsächlich um das HTTP-Protokoll, dennoch muss erwähnt werden, dass curl darüber hinaus zahlreiche andere Protokolle, unter anderem das File Transfer Protocol (FTP) und Post Office Protocol Version 3 (POP3), unterstützt [50].

Zum händischen Testen der Endpunkte für diese Arbeit wird aufgrund der einfacheren Bedienung und persönlicher Präferenz Postman verwendet.

3 Festlegung der Testbedingungen

Isabell Waas

Nachdem die wichtigsten Grundbegriffe und -Prinzipien erläutert wurden, sollen die Testbedingungen beleuchtet werden, unter denen die geeignete Kombination aus Programmiersprache und Framework zur Entwicklung von RESTful Webservices ermittelt werden soll.

3.1 Auswahl der Programmiersprachen und Frameworks

Steven Solledder

Es gibt zahlreiche Programmiersprachen und Frameworks, welche im Rahmen dieser Arbeit untersucht werden könnten. Zum einen soll diese Arbeit Erkenntnisse bringen, welches Framework zur Erstellung eines RESTful-Webservices am besten geeignet ist. Zum anderen sollen diese Erkenntnisse im Idealfall so vielen Entwicklern wie möglich von Nutzen sein.

Um die Popularität einer Programmiersprache festzustellen, gibt es viele Möglichkeiten. Da es zur Softwareentwicklung vor allem online Ressourcen gibt, darunter zum Beispiel Blogs, Dokumentationen oder Papers, und die große Mehrheit der Menschen Google nutzt [49], wurde zur Bestimmung der Popularität der PYPL-Index verwendet. Dieser listet die Beliebtheit von Programmiersprachen auf Basis der Rohdaten von Google Trends [3]. Dabei wurden alle Programmiersprachen ausgeschlossen, welche nicht statisch typisiert sind oder bei welchen sich manuell um das Speichermanagement gekümmert werden muss.

Es werden nur statisch typisierte Programmiersprachen berücksichtigt, da diese vor allem für größere Projekte oder Projekte, in welchen unterschiedliche Menschen an verschiedenen Teilen arbeiten, unter anderem aus den folgenden Gründen besser geeignet sind. Zunächst werden durch die im Code hinterlegten Typinformationen bereits zur Compile Time einige Fehler erkannt, welche ansonsten gar nicht oder nur durch Suchen gefunden werden. Dies führt auch dazu, dass die Software weniger fehleranfällig ist. Ein weiterer Grund für statisch typisierte Sprachen ist, dass integrierte Entwicklungsumgebungen in vielerlei Hinsicht wesentlich besser bei der Programmierung helfen können. So zeigen sie sinnvollere Vorschläge an oder können Code leichter refaktorisieren. Ebenfalls von Vorteil ist, dass Compiler oder Transpiler aufgrund der Typinformationen den Code besser optimieren können, was in den meisten Fällen zu einer höheren Performance führt.

Programmiersprachen mit manuellem Speichermanagement wie zum Beispiel C oder C++ werden hingegen nicht berücksichtigt, da das händische Kümmern um Allokation und Freigabe von Speicher leichter zu einem Buffer Overflow und Memory Access Violation und somit leichter zu Sicherheitslücken führen kann.

Abgesehen davon gibt es noch zahlreiche weitere Gründe, die für statisch typisierte Programmiersprachen ohne manuellem Speichermanagement sprechen, die jedoch nicht im Rahmen dieser Arbeit liegen. Ausgehend von den genannten Auswahlkriterien sind aus dem PYPL-Index folgende Sprachen interessant, die in der Abbildung unten grün markiert sind.

Worldwide, Feb 2023 compared to a year ago:				
Rank	Change	Language	Share	Trend
1		Python	27.7 %	-0.7 %
2		Java	16.79 %	-1.3 %
3		JavaScript	9.65 %	+0.6 %
4	↑	C#	6.97 %	-0.5 %
5	↓	C/C++	6.87 %	-0.6 %
6		PHP	5.23 %	-0.8 %
7		R	4.11 %	-0.1 %
8	↑↑	TypeScript	2.83 %	+0.8 %
9		Swift	2.27 %	+0.3 %
10	↓↓	Objective-C	2.25 %	-0.1 %
11	↑↑	Go	1.95 %	+0.7 %
12	↑↑	Rust	1.91 %	+0.9 %
13	↓	Kotlin	1.85 %	+0.2 %
14	↓↓↓	Matlab	1.71 %	-0.1 %
15	↑	Ruby	1.11 %	+0.3 %
16	↓	VBA	1.03 %	+0.0 %
17		Ada	0.94 %	+0.3 %
18	↑↑	Dart	0.84 %	+0.4 %
19		Scala	0.62 %	+0.0 %
20	↑↑↑	Lua	0.54 %	+0.2 %

Abbildung 5: Erste 20 Plätze des PYPL-Index [3]

Da neben der generellen Popularität auch die Nachfrage am deutschen Arbeitsmarkt mit einbezogen werden soll, werden in folgender Tabelle zu den grün markierten Sprachen die Anzahl der gesuchten Jobs, welche bei der staatlichen Bundesagentur für Arbeit gemeldet sind, zugeordnet.

Programmiersprache	Jobangebote
C#	25779
Java	4657
Go	3367
TypeScript	873
Rust	384
Kotlin	279
Swift	153
Scala	126
Dart	100

Tabelle 1: Jobangebote der Bundesagentur für Arbeit für ausgewählte Programmiersprachen [49]

Von den abgebildeten Sprachen sollen im Rahmen dieser Arbeit die obersten vier näher betrachtet werden, um den Umfang einzuschränken.

Doch die Programmiersprache stellt nur die eine Hälfte dar, denn es muss für jede Sprache auch noch ein passendes Framework gesucht werden. Hierzu soll, ähnlich wie bei den Programmiersprachen, jeweils das Populärste gesucht werden. Dafür wird auf die größte Codebibliothek der Welt, GitHub¹¹, zurückgegriffen, nach dem Suchbegriff „framework“ gesucht und für jede der vier Sprachen ermittelt, welches das auf Basis der vergebenen Sterne beliebteste Framework für RESTful Webservice ist. Natürlich gibt es auch hier wieder verschiedene Ansätze, dies ausfindig zu machen, aufgrund des großen Marktanteils von GitHub erscheint das gerade erläuterte Vorgehen jedoch am besten geeignet zu sein [16]. Da TypeScript¹², wie in Kapitel 4.3.1 noch genauer erläutert wird, nur zu JavaScript transpiliert wird und es zu allen großen JavaScript-Frameworks entweder eingebaute oder separat herunterladbare Typen gibt, wurde im Falle von TypeScript in GitHub als Sprachfilter „JavaScript“ festgelegt. Ansonsten würde das Ergebnis verfälscht werden, da fast kein JavaScript-Framework unter dem Sprachfilter „TypeScript“ gelistet ist.

¹¹<https://github.com/>

¹²<https://www.typescriptlang.org/>

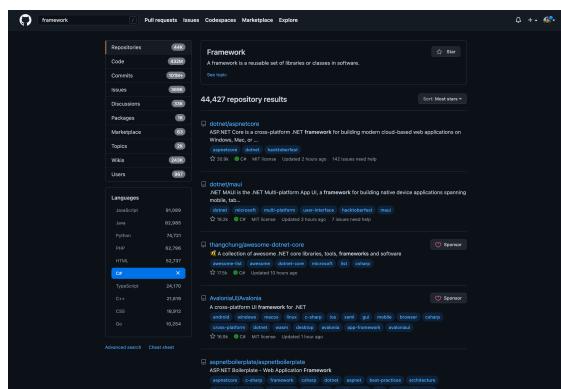


Abbildung 6: Frameworks für C# [8]

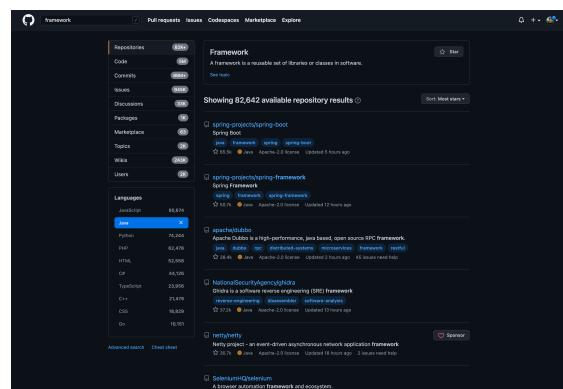


Abbildung 7: Frameworks für Java [8]

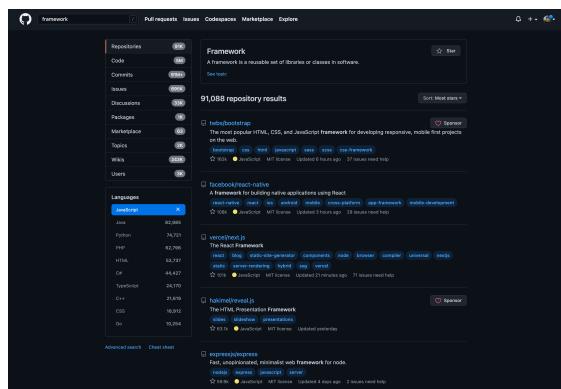


Abbildung 8: Frameworks für JavaScript [8]

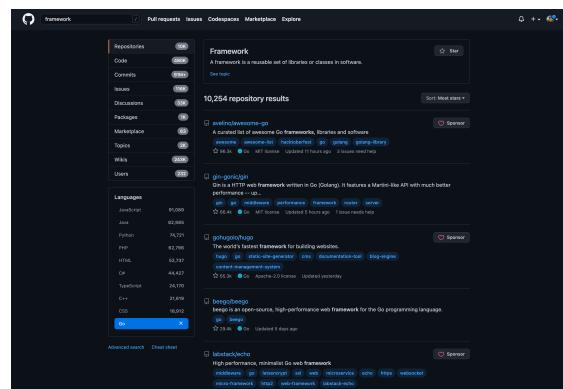


Abbildung 9: Frameworks für Go [8]

Den abgebildeten Bildschirmfotos von GitHub kann entnommen werden, dass das jeweils beliebteste für RESTful Webservices geeignete Framework jeder Sprache das in der unten stehenden Tabelle genannte ist. Damit stehen die Sprachen und Frameworks fest:

Programmiersprache	Framework
C#	ASP.NET Core
Java	Spring Boot
Go	Gin
TypeScript	Express.js

Tabelle 2: Ausgewählte Sprachen und Frameworks

Abschließend soll nochmal erwähnt werden, dass es auch andere Vorgehensweisen zur Auswahl gibt, aus genannten Gründen jedoch die beschriebene sinnvoll erscheint.

3.2 Aufgaben

Steven Solleeder

Nun stehen die Programmiersprachen und Frameworks fest und es kann auf die Aufgaben eingegangen werden, welche in den im Rahmen dieser Arbeit entstehenden Webservices umgesetzt werden sollen. Diese dienen später als Basis für die Bewertung jedes einzelnen Frameworks anhand mehrerer Kriterien (siehe Kapitel 3.3).

Zur Veranschaulichung der typischen Anforderungen eines RESTful Webservices, wurde eine Spezifikation mit Swagger erstellt. Letzteres ist eine Sammlung verschiedener Regeln und Werkzeuge, um eine standardisierte OpenAPI Spezifikation zu erstellen [42]. Swagger wird zum Beschreiben der Struktur von APIs verwendet. Für die vorliegende Arbeit wurde eine beispielhafte Bibliotheks-API namens „Library API“ definiert, in welcher Bücher verwaltet werden können.

3.2.1 Definition von Ressourcen und Data Transfer Objects

Isabell Waas

```

Schemas ^

Book ▼ {
  id          integer
  minimum: 0
  isbn        string
  pattern: ^(978-?|979-?)\d{1,5}-?\d{1,7}-?\d{1,6}-?\d{1,3}$
  title       string
  minLength: 1
  maxLength: 300
  author      string
  minLength: 1
  maxLength: 100
}

ErrorResponse ▼ {
  message     string
  minLength: 1
}

ErrorsResponse ▼ {
  messages    ▼ [string]
}

BooksResponse ▼ {
  maximumPage integer
  minimum: 1
  items       ▼ [Book ▼ {
    id          integer
    minimum: 0
    isbn        string
    pattern: ^(978-?|979-?)\d{1,5}-?\d{1,7}-?\d{1,6}-?\d{1,3}$
    title       string
    minLength: 1
    maxLength: 300
    author      string
    minLength: 1
    maxLength: 100
  }]
}

IdResponse ▼ {
  id          integer
  minimum: 0
}

```

Abbildung 10: Spezifikation der Schemata

Die zentralen Bestandteile eines RESTful Webservices sind die Ressourcen. Im Falle der API zur Bibliotheksverwaltung ist zum Beispiel ein Buch eine Ressource. Diese soll anschließend über verschiedene Endpunkte erstellt, gelesen, aktualisiert und gelöscht werden. Bei der Definition des Buches sind die Eigenschaften zu definieren, welche jedes Objekt des Typs Buch besitzen soll. In Abbildung 10 ist zu sehen, welche Merkmale in diesem Projekt

betrachtet werden. Selbstverständlich hätte ein Buch in einer realen API wesentlich mehr Eigenschaften, jedoch sind die dargestellten für die Veranschaulichung der typischen Aufgaben eines Webservices vollkommen ausreichend.

Abgesehen von der Ressource Buch zeigt die Spezifikation in Abbildung 10 noch weitere Schemata, die die Antworten des Webservices auf verschiedene Anfragen beschreiben. Sie werden auch als Data Transfer Objects (DTOs) bezeichnet.

Die ErrorResponse oder ErrorsResponse werden immer dann verwendet, wenn ein Statuscode auftritt, welcher einen Fehler anzeigt. Das sind alle Statuscodes, die mit der Ziffer 4 oder 5 beginnen. In so einem Fall soll die Response des jeweiligen Endpunkts genauere Informationen liefern, warum es zu einem Fehler kam. Diese Erklärung befindet sich bei der ErrorResponse in einem Attribut message. Traten mehrere Fehler auf, die einander nicht beeinflussen, wird die ErrorsResponse zurückgegeben, welche ein Attribut messages des Typs Array enthält, wodurch mehrere Fehlernachrichten in der Antwort des Webservices enthalten sein können.

Daneben gibt es noch die DTOs IdResponse und BooksResponse. Was Ersteres betrifft, so handelt es sich um ein einfaches Objekt, das die Id einer Ressource beinhaltet. Zu dem zweiten Schema muss gesagt werden, dass es in RESTful Webservices manchmal vorkommt, dass nicht nur eine oder mehrere Ressourcen in der Response zurückgegeben werden, sondern noch weitere Informationen. Welche Informationen dies im Falle der BooksResponse genau sind, wird in Kapitel 3.2.3 erläutert, da dieses Schema in dem im genannten Kapitel erläuterten Endpunkt verwendet wird.

3.2.2 Abruf eines Objekts

Isabell Waas

The screenshot shows the API specification for the endpoint `/v1/books/{id}`. The endpoint is described as "Get a book by its id.".

Parameters:

Name	Description
id * required integer (path)	The id of the book to retrieve. <input type="text" value="id"/>

Responses:

Code	Description	Links
200	OK	No links
404	Not Found	No links
500	Internal Server Error	No links

Example Value | Schema

200 OK Response Schema:

```
Book < v {
    id      integer
            minimum: 0
    isbn    string
            pattern: ^(978-?|979-?)?\\d{1,5}-?\\d{1,7}-?\\d{1,6}-?\\d{1,3}$
    title   string
            minLength: 1
            maxLength: 300
    author  string
            minLength: 1
            maxLength: 100
}
```

404 Not Found Response Schema:

```
ErrorResponse < v {
    message string
            minLength: 1
}
```

500 Internal Server Error Response Schema:

```
ErrorResponse < v {
    message string
            minLength: 1
}
```

Abbildung 11: Spezifikation des Endpunkts zum Abrufen eines Buches

Eine der typischsten Aufgaben eines Webservices mit der REST-Architektur ist das Abrufen einer einzelnen Ressource, bei dem beschriebenen Webservice eines Buches. Um das richtige Buch zu finden, benötigt es einen eindeutigen Schlüssel, der es von den anderen Büchern unterscheidet. Dieser Schlüssel ist nahezu immer eine Id. Er wird in der URL, unter der die Ressource erreichbar ist, mit angegeben, wie auch Abbildung 11 zeigt. Bei dem Abruf der Ressource muss zudem das HTTP-Verb GET genutzt werden, welches spezifiziert, dass Daten gelesen werden sollen.

Die HTTP-Response beinhaltet schließlich im Erfolgsfall den Statuscode 200 sowie das abgefragte Buch. Wurde kein Buch mit der gewünschten Id gefunden, so erhält der Client als Antwort den Statuscode 404 und eine ErrorResponse, welche ihm mitteilt, worin das Problem besteht. Im Falle eines anderen Fehlers sendet der Server den Statuscode 500 und eine ErrorResponse zurück.

3.2.3 Abruf aller Objekte mit Pagination, Sortierung und Filterung

Isabell Waas

The screenshot shows the Swagger UI interface for a REST API endpoint. At the top, it displays a blue button labeled "GET" and the URL "/v1/books". To the right of the URL is a brief description: "Get all books ordered by id and optionally filtered." Below this is a "Parameters" section containing three fields:

- query**: A "string" parameter with a description "Term to look for in all attributes." and a text input field containing "query".
- page * required**: An "integer" parameter with a description "Number of page to retrieve." and a text input field containing "page".
- per_page * required**: An "integer" parameter with a description "Number of records per page." and a text input field containing "per_page".

Below the parameters is a "Responses" section with three entries:

- 200 OK**: The response status is "OK". The "Media type" dropdown is set to "application/json". The "Example Value" and "Schema" sections both show the JSON schema for a "BooksResponse" object, which contains an array of "Book" objects. Each "Book" object has properties: "id" (integer, min: 1), "isbn" (string), "title" (string, min: 1, max: 300), and "author" (string, min: 1, max: 100).
- 400 Bad Request**: The response status is "Bad Request". The "Media type" dropdown is set to "application/json". The "Example Value" and "Schema" sections both show the JSON schema for an "ErrorsResponse" object, which contains an array of strings under the "messages" key.
- 500 Internal Server Error**: The response status is "Internal Server Error". The "Media type" dropdown is set to "application/json". The "Example Value" and "Schema" sections both show the JSON schema for an "ErrorResponse" object, which contains a single string under the "message" key.

Abbildung 12: Spezifikation des Endpunkts zum Abrufen aller Bücher

Auch das Abrufen aller Objekte einer Ressource wird in fast jedem RESTful Webservice durch einen Endpunkt realisiert, weshalb auch in den Webservices der vorliegenden Arbeit ein solcher existieren soll. Er ist auf der vorherigen Seite abgebildet und liefert alle Bücher aufsteigend sortiert nach ihrer Id. Bei seinem Aufruf muss das HTTP-Verb GET angegeben werden. Für den Endpunkt gibt es zudem mehrere, teilweise optionale Query-Parameter. Über den Parameter query kann ein Suchbegriff angeführt werden, welcher auf alle Attribute des Buches angewandt wird. Das bedeutet, der Begriff wird sowohl im Titel als auch in Autor und ISBN gesucht. Da allerdings mindestens genauso häufig alle Bücher ungefiltert benötigt werden, ist der Parameter query optional.

Nicht selten verwenden RESTful Webservices eine sehr umfassende Datenbasis. Das Ergebnis eines Endpunkts zum Abrufen aller Objekte einer Ressource besteht in so einem Fall oft aus einer großen Anzahl an Einträgen. Um den Umgang mit der Antwort eines solchen Endpunkts einfacher zu gestalten, wird meistens Pagination eingesetzt. Das bedeutet, dass die gesamte Ergebnismenge in kleinere Teile, die sogenannten pages (englisch für „Seiten“), aufgeteilt wird. Hierfür gibt es im Endpunkt aus Abbildung 12 die Query-Parameter page und per_page. Letzterer bestimmt, wie viele Einträge jede page beinhaltet. Dabei ist zu beachten, dass insofern die gesamte Ergebnismenge nicht restlos durch die per_page-Anzahl geteilt werden kann, auf der letzten Seite weniger Einträge sein können. Der Parameter page bestimmt, der wievielte Teil an Ergebnissen abgerufen werden soll. Da bei einer Bibliothek davon auszugehen ist, dass es selbst mit Filterung durch einen Suchbegriff eine große Anzahl an Büchern gibt, wird in dem Endpunkt der Library API immer Pagination eingesetzt. Die Query-Parameter page und per_page sind folglich zwingend anzugeben.

Was die Response betrifft, so muss selbstverständlich eine Menge an Büchern zurückgesendet werden. Darüber hinaus soll jedoch auch die Information mitgeliefert werden, welche page mit der per_page-Anzahl aus dem Request maximal abgerufen werden kann, sodass sie noch Ergebnisse beinhaltet. Aus diesem Grund wird im Erfolgsfall der Statuscode 200 und das bereits in Kapitel 3.2.1 angesprochene Schema BooksResponse verwendet. Dieses beinhaltet die Eigenschaften items, welche die Menge an Büchern enthält, sowie maximumPage. Waren nicht alle Query-Parameter korrekt, hat die Antwort den Statuscode 400 und liefert eine ErrorsResponse. Bei anderen Fehlern lautet der Statuscode 500 und es wird ebenfalls eine ErrorResponse zurückgesendet.

3.2.4 Erstellung eines Objekts

Steven Solleder

The screenshot shows the API specification for the endpoint `/v1/books`. The endpoint is described as "Create a new book".

Parameters: No parameters.

Request body (required): application/json

The request body schema is defined as follows:

```

    {
      isbn*: string
        pattern: ^(978-?|979-?)?\d{1,5}-?\d{1,7}-?\d{1,6}-?\d{1,3}$
      title*: string
        minLength: 1
        maxLength: 300
      author*: string
        minLength: 1
        maxLength: 100
    }
  
```

Responses:

Code	Description	Links
201	Created	No links
400	Bad Request	No links
500	Internal Server Error	No links

201 Created:

Media type: application/json

The response schema is defined as follows:

```

  IdResponse {
    id: integer
      minimum: 0
  }
  
```

400 Bad Request:

Media type: application/json

The response schema is defined as follows:

```

  ErrorsResponse {
    messages: [string]
  }
  
```

500 Internal Server Error:

Media type: application/json

The response schema is defined as follows:

```

  ErrorResponse {
    message: string
      minLength: 1
  }
  
```

Abbildung 13: Spezifikation des Endpunkts zum Erstellen eines Buches

Damit Objekte einer Ressource abgerufen werden können, muss es natürlich auch eine Möglichkeit geben, vorher Objekte anzulegen. Hierfür wird ein Endpunkt verwendet, welcher das HTTP-Verb POST nutzt. In der Library API sieht der Endpunkt zum Erstellen eines Buches wie in Abbildung 13 aus. Zu beachten ist, dass im Request-Body alle Eigenschaften des Buches geschickt werden müssen, außer die Id. Dies ist dadurch begründet, dass Ids in der Regel von der Datenbasis, in den meisten Fällen einer Datenbank, einem Objekt bei dessen Erstellung zugewiesen werden. Sie werden dabei automatisch inkrementiert, wodurch sichergestellt wird, dass keine Id mehrfach vorkommt.

Konnte das Objekt erstellt werden, so beinhaltet die Antwort den Statuscode 201 und die Id des Objektes. Waren die im Request-Body gesendeten Daten unvollständig oder fehlerhaft, so erhält der Client auf seine Anfragen hin den Statuscode 400 und eine ErrorsResponse. Tritt ein anderer Fehler auf, so lautet der Statuscode wieder 500 und es wird eine ErrorResponse zurückgegeben.

3.2.5 Änderung einzelner Attribute eines Objekts

Steven Solleder

The screenshot shows a REST API documentation interface for the `PATCH /v1/books/{id}` endpoint. The endpoint is described as "Update a book by its id.".

Parameters:

- Name:** `id` **Description:** The id of the book to update. **Type:** integer **(path)**. A text input field contains the value `id`.

Request body required: application/json

The data necessary to update a book.

Responses:

Code	Description	Links
204	No Content	No links
400	Bad Request	No links
404	Not Found	No links
500	Internal Server Error	No links

Example Value | Schema:

```

{
  isbn: string,
  title: string,
  author: string
}
  
```

Example Value | Schema:

```

ErrorResponse {
  messages: [string]
}
  
```

Example Value | Schema:

```

ErrorResponse {
  message: string
}
  
```

Abbildung 14: Spezifikation des Endpunkts zum Aktualisieren eines Buches

Bei vielen Ressourcen eines RESTful Webservices ist es möglich, dass sich bestimmte Eigenschaften ändern. So kann zum Beispiel eine Person durch eine Heirat ihren Nachnamen wechseln, ein Gegenstand einen neuen Anstrich erhalten oder der Status einer Bestellung aktualisiert werden. Im Falle der Merkmale, die die Ressource Book besitzt, sind derartige Änderungen zwar eher unwahrscheinlich, jedoch könnte z. B. bei der Erstellung eines Buches ein Schreibfehler unterlaufen sein. In diesem Fall ist ein Update dieses Objektes dennoch sinnvoll. Hierfür gibt es einen Endpunkt, welcher das gewünschte Buch mithilfe seiner Id findet und über das HTTP-Verb PATCH funktioniert. PATCH wird dafür verwendet, nur ausgewählte Eigenschaften einer Ressource zu updaten und nicht die gesamte Ressource. Der Request-Body kann daher die ISBN, den Titel und den Autor enthalten, jedoch ist jeder der Werte optional.

War die Aktualisierung erfolgreich, so wird mit dem Statuscode 204 geantwortet. Waren die gesendeten Daten fehlerhaft, so erhält der Client eine ErrorResponse und den Statuscode 400. Wurde kein Buch mit der angegebenen Id gefunden, so wird der Statuscode 404 und eine ErrorResponse zurückgeschickt. Auch bei diesem Endpunkt beinhaltet die Antwort im Falle eines anderen Fehlers den Statuscode 500 und eine ErrorResponse.

3.2.6 Löschung eines Objekts

Isabell Waas

DELETE /v1/books/{id} Delete a book by its id.

Parameters

Name	Description
id * required integer (query)	Id of the book to delete. id

Responses

Code	Description	Links
204	No Content	No links
404	Not Found	No links
500	Internal Server Error	No links

Media type: application/json

Example Value | Schema

```
ErrorResponse <-
  message: string
    minLength: 1
```

Abbildung 15: Spezifikation des Endpunkts zum Löschen eines Buches

So wie ein bestimmtes Objekt über seine Id abgerufen und aktualisiert werden kann, so kann es über seinen individuellen Schlüssel auch gelöscht werden. Hierfür kommt das HTTP-Verb DELETE zum Einsatz. Der Endpunkt zum Löschen eines Buches ist oben dargestellt.

Die Antwort des Servers ist nach einem erfolgreichen Löschkvorgang leer und beinhaltet lediglich den Statuscode 204. Wurde die gewünschte Ressource nicht gefunden, so werden wieder der Statuscode 404 und eine entsprechende ErrorResponse zurückgeschickt. In allen anderen Fehlerfällen erhält der Client den Statuscode 500 und eine ErrorResponse.

Damit wurden alle in den RESTful Webservices, welche für diese Arbeit implementiert werden, verwendeten Endpunkte erläutert. Die nachfolgende Grafik zeigt alle Endpunkte der Library API sowie die dabei verwendeten Schemata nochmal in einem kurzen Überblick.

Library API 1.0.0 OAS3

[specs.yaml](#)

The Library API is intended to illustrate typical requirements of RESTful web services. It can be used to manage various books.

Books

Endpoints concerning books

- POST** /v1/books Create a new book.
- GET** /v1/books Get all books ordered by id and optionally filtered.
- GET** /v1/books/{id} Get a book by its id.
- PATCH** /v1/books/{id} Update a book by its id.
- DELETE** /v1/books/{id} Delete a book by its id.

Schemas

- [Book >](#)
- [ErrorResponse >](#)
- [ErrorsResponse >](#)
- [BooksResponse >](#)
- [IdResponse >](#)

Abbildung 16: Übersicht der Spezifikation

3.2.7 Validierung von erhaltenen Daten

Steven Solleder

Das Validieren ist eine der wichtigsten Aufgaben eines jeden RESTful Webservices. So kann der Client im Falle falsch gesendeter Daten über diesen Fehler benachrichtigt werden, wodurch er zugleich darauf aufmerksam gemacht wird, dass die Daten erneut geschickt werden müssen. Auch weiß der Server bei einer erfolgreichen Validierung, dass die angekommenen Daten im erwarteten Zustand vorliegen und nun entsprechend weiterverarbeitet werden können.

Das Validieren von Daten einkommender Anfragen bedeutet diese sowohl auf ihre Form als auch den konkreten Inhalt hin zu überprüfen. Formvalidierung heißt dabei, dass ein gesendetes Objekt sowohl die erwarteten Attribute besitzt als auch, dass die Attributwerte von einem bestimmten Typ sind. Inhaltsvalidierung besagt, dass ein Attributwert bestimmten Anforderungen genügen muss, z. B. dass ein String mindestens drei und maximal 10 Zeichen lang ist.

Jeder im Rahmen dieser Arbeit implementierte Endpunkt validiert seine Daten. Zur Einhaltung dieser Aufgabe soll zum einen die Validierung korrekt sein, zum anderen soll ein Endpunkt im Fehlerfall den Statuscode 400 und ein Objekt des Typs ErrorsResponse schicken. Die einzelnen Nachrichten des „messages“-Arrays müssen dabei nur im Wesentlichen den aufgekommenen Fehler wiedergeben.

3.2.8 Behandlung von Fehlern

Steven Solleder

Beim Aufrufen eines Endpunktes können zahlreiche Fehler auftreten. Dazu gehören auf der einen Seite Fehler, an welche bei der Entwicklung gedacht wurde und für die, insofern möglich, eine sinnvolle Fehlerbehandlung durchgeführt wird. Auf der anderen Seite gibt es Fehler, an die bei der Entwicklung nicht gedacht wurde. Deren Ursache ist nicht selten, dass einige Programmiersprachen nur oder zumindest zu einem Teil Checked Exceptions unterstützen, was zur Folge hat, dass der Compiler nicht explizit überprüft, dass alle auftreffenden Fehler behandelt werden. Auch können solche Fehler durch fehlende Entwicklungszeit oder unerfahrene Entwickler entstehen. In jedem Fall kann es geworfene Fehler

geben, welche sich ihren Weg bis zum Startpunkt der Anwendung bahnen. Diese dürfen natürlich nicht zum Absturz der Anwendung führen.

Damit diese Aufgabe erfüllt ist, muss es einen universellen Mechanismus geben, um nicht speziell behandelte Fehler dennoch abzufangen. Außerdem darf solch ein Fehler nicht zum Absturz des RESTful Webservices oder zur Einschränkung anderer gleichzeitig stattfindender Aufrufe führen, sondern nur zur Beendigung des Aufrufs des aktuellen Endpunktes. Ebenso muss dem Aufrufer des Endpunktes eine sinnvolle Antwort gesendet werden. Auch soll es einfach möglich sein, diesen universellen Mechanismus bewusst auszulösen.

3.2.9 Konfiguration des Servers

Steven Solleder

Jeder Server muss bis zu einem bestimmten Punkt konfiguriert werden. Dazu gehören zum Beispiel Zugangs- und Adressdaten zu verwendeten Datenbanken oder der Port, unter welchem der Server erreichbar ist. Indem die Konfiguration von dem eigentlichen Server getrennt ist, ist es somit möglich, Teile des Servers zu ändern, welche häufiger oder wahrscheinlicher Änderungen ausgesetzt sind, ohne den Anwendungscode zu modifizieren. Das führt dazu, dass diese Teile sehr schnell angepasst werden können. Außerdem hat es zur Folge, dass der Server keinen erneuten Tests unterzogen werden muss, um zu garantieren, dass er noch in Anbetracht seiner Logik wie vorhergesehen funktioniert.

Im Rahmen dieser Aufgabe soll ein einheitlicher Ort zur Konfiguration des Servers in Form einer oder mehrerer Dateien entwickelt werden. Dabei ist es zum Beispiel möglich, dass die Konfiguration möglichst flexibel getroffen werden kann. Das kann bedeuten, dass es mehrere Konfigurationen gibt, welche voneinander erben können. Ebenso vorstellbar wären Bedingungen und Variablen innerhalb der Konfigurationsdatei.

3.3 Bewertungskriterien

Steven Solleder

Nachdem nun alle Aufgaben näher erläutert wurden, müssen diese bewertet werden. Dazu werden die folgenden Bewertungskriterien herangezogen, wobei erklärt wird, warum diese sinnvoll sind und wie das nähere Vorgehen der Bewertung stattfindet. Dabei gibt es sowohl qualitative als auch quantitative Bewertungskriterien. Selbstverständlich werden alle Frameworks mit allen Bewertungskriterien auf gleiche Weise geprüft, sodass die Ermittlung des besten Frameworks gerecht abläuft.

3.3.1 Dokumentation und Wartung

Isabell Waas

Soll ein neues Framework zum Einsatz kommen, so stellt für die meisten Entwickler dessen offizielle Dokumentation den ersten Anlaufpunkt dar. Diese ist im besten Fall sehr umfassend und leicht verständlich, im schlimmsten Fall jedoch sehr unübersichtlich und oberflächlich. Trifft Letzteres zu, so muss in der Regel auf andere, weniger vertrauenswürdige Quellen zugegriffen werden und vieles durch Ausprobieren erschlossen werden. Häufig wird dann der beste Weg, eine bestimmte Anforderung umzusetzen, nicht gefunden.

In Bezug auf die Frameworks, welche im Rahmen dieser Arbeit verwendet werden, soll daher betrachtet werden, wie gut die Dokumentation ist. Bei dieser Beurteilung handelt es sich selbstverständlich um eine qualitative Untersuchung, da keine objektiven Analysemethoden existieren, um die Qualität einer Dokumentation zu bewerten. Aus eigener Erfahrung kann jedoch gesagt werden, dass eine vollständige und in die Tiefe gehende Dokumentation viel Beispielcode sowie Tutorials enthält. Auch sollte sie übersichtlich sein. Des Weiteren werden Klassen und Methoden ausführlich beschrieben und zu jedem für das jeweilige Framework relevanten Thema sind alle wichtigen Informationen schnell auffindbar und nicht über mehrere Webseiten verteilt. Auf externe Quellen sollte so wenig wie möglich zugegriffen werden müssen.

Abgesehen davon ist auch die Wartung eines Frameworks nicht zu vernachlässigen. Das bedeutet insbesondere, wie häufig ein neuer Release veröffentlicht wird. Diese quantitative Untersuchung soll ebenfalls in dieser Arbeit vorgenommen werden.

3.3.2 Lines of Code

Isabell Waas

Das wohl umstrittenste Kriterium, welches in dieser Arbeit betrachtet werden soll, ist die als Lines of Code (LOC) bezeichnete Metrik. Dabei handelt es sich um die Anzahl der Zeilen an Quelltext einer Software. Sie steht häufig dafür in der Kritik, dass von einer besonders großen oder kleinen Zeilenanzahl weder darauf geschlossen werden kann, dass der Code besonders effizient ist, noch auf das Gegenteil. Denn wie viel Code produziert wird, wird durch zahlreiche Aspekte beeinflusst. So sind zum Beispiel das genutzte Framework und die Architektur des Projektes relevant, aber auch die Kenntnisse des Programmierers über bestimmte Sprachfeatures sowie die Formatierung des Quelltextes. Alleine die Tatsache, ob der Entwickler die in zahlreichen Programmiersprachen benötigten geschweiften Klammern in dieselbe Zeile wie die vorherige Anweisung oder in eine separate Zeile schreibt, kann die Menge des entstehenden Codes stark verändern. Außerdem kann es immer sein, dass eine Software sehr ineffizient und redundant programmiert ist, während die andere von einem erfahreneren Entwicklungsteam stammt, das wesentlich besseren Code schreibt. Dennoch hat die LOC Metrik auch ihren Nutzen. Insbesondere kann sie einen ersten Eindruck vermitteln, wie komplex eine Software im Vergleich zu anderen Projekten ist. Insgesamt kann die Metrik, wenn die genannten Probleme im Hinterkopf behalten und im besten Fall noch weitere Bewertungskriterien herangezogen werden, trotzdem verwendet werden [39].

Bei der Softwareentwicklung werden typischerweise häufig leere Zeilen verwendet, um den Quelltext lesbarer zu machen und besser zu strukturieren. Ebenso sind an manchen Stellen Kommentare zu finden, die wichtige Gedankengänge festhalten oder komplexe Codestellen erläutern. Um eine realitätsgereitere Anzahl an Codezeilen zu erhalten, welche tatsächlich für die Software relevante Inhalte haben, wurden verschiedene Berechnungsarten entwickelt. Während die klassische LOC-Berechnung auch leere Zeilen und Kommentare mit einberechnet, werden diese bei der Source Lines of Code (SLOC) Metrik nicht berücksichtigt. Darüber hinaus gibt es die Logical Lines of Code (LLOC) Metrik, die ebenfalls keine Kommentare und Leerzeilen berechnet. Im Gegensatz zu SLOC zählt sie jedoch nicht die Codezeilen, sondern die Anweisungen [39]. Mithilfe des folgenden Listings in der Programmiersprache Java soll der Unterschied zwischen SLOC und LLOC näher erklärt werden.

```

1 class Main
2 {
3     public static void main(String [] args)
4     {
5         for (int i = 0; i<10; i++) System.out.println("hello world");
6     }
7 }
```

Listing 3: Codebeispiel zur Veranschaulichung der SLOC und LLOC Metriken

In dem Codebeispiel wird eine for-Schleife genutzt, welche eine einzige Anweisung enthält und daher in eine Zeile geschrieben werden kann. Führt man die Messung mit SLOC durch, so würde man eine einzige Codezeile zählen, da SLOC die physischen Zeilen betrachtet. Die LLOC Metrik berechnet hingegen zwei logische Codezeilen, da die Definition der for-Schleife und die print-Anweisung innerhalb der Schleife zwei einzelne logische Befehle sind [51]. Folglich liefert die LLOC Metrik die zuverlässigste Anzahl an Codezeilen, da sie die Formatierung des jeweiligen Entwicklers weitestgehend vernachlässigt.

Eines der am häufigsten für das Generieren von Code Metriken genutzte Werkzeug ist Count Lines of Code (cloc)¹³. Das in Perl verfasste Kommandozeilentool nutzt jedoch zur Berechnung der Codezeilen SLOC. Da kein ebenso etabliertes Tool gefunden wurde, welches die LLOC Metrik verwendet und alle benötigten Programmiersprachen unterstützt, soll im Rahmen dieser Arbeit cloc eingesetzt werden. Jedoch wird jede logische Anweisung in eine eigene physische Zeile geschrieben, was besser lesbar ist und dafür sorgt, dass mit der SLOC Metrik dieselbe Zeilenanzahl berechnet wird, welche sich auch mit LLOC ergeben würde. Zur Einheitlichkeit der Messungen für die einzelnen Frameworks, soll zudem auf einen ähnlichen Codestil geachtet werden, indem z. B. jede öffnende geschweifte Klammer in dieselbe Zeile mit der vorherigen Anweisung geschrieben wird. Mit cloc soll schließlich für jeden der jeweils in einem anderen Framework entstandenen Webservices die für alle in Kapitel 3.2 beschriebenen Aufgaben insgesamt benötigte Anzahl an Codezeilen quantitativ ermittelt und bewertet werden. Dabei werden Konfigurationsdateien, die aus Konstanten beziehungsweise Key-Value-Pairs bestehen, nicht berücksichtigt.

¹³<https://github.com/AlDanial/cloc>

3.3.3 Notwendigkeit zusätzlicher Bibliotheken

Steven Solleeder

In der Softwareentwicklung dienen Bibliotheken dazu, einmal entwickelte Funktionalität kostenlos oder entgeltlich weiterzugeben. Dies hat den wesentlichen und offensichtlichen Vorteil, dass die gleiche oder ähnliche Funktionalität nicht immer wieder aufs Neue geschrieben werden muss. Das spart Zeit und Geld. Dementsprechend ist es zwar prinzipiell sinnvoll, Software auf Basis verschiedener Bibliotheken aufzubauen, jedoch kommen mit der Nutzung verschiedener Bibliotheken auch Probleme einher.

Zunächst stehen Bibliotheken meistens für sich, was dazu führt, dass verschiedene Bibliotheken nicht mit den Klassen und Funktionen anderer Bibliotheken umzugehen wissen. Um sich diesem Problem entgegenzustellen, hilft häufig nur das Anwenden von Design-Patterns.

Ein passendes Beispiel dafür wäre das Adapter-Pattern.

Ein weiterer Nachteil ist, dass es häufig für einen Zweck zahlreiche verschiedene Bibliotheken gibt, welche erst miteinander verglichen und auf Eignung geprüft werden müssen.

Ebenso kann es zu Problemen kommen, wenn die eigene Software auf einer Bibliothek aufbaut, welche seit Längerem nicht mehr aktualisiert wurde, da das gesamte Projekt dann oft nicht auf eine neuere Version der verwendeten Sprache beziehungsweise Laufzeitumgebung aktualisiert werden kann. Dies trifft natürlich nur zu, wenn die Sprache oder die Laufzeitumgebung, insofern vorhanden, so konzipiert sind, dass bei einer neuen Version alter Code, Bytecode oder Maschinencode aufgrund von Breaking Changes nicht mehr ausgeführt werden können.

Aus den genannten Gründen wird bei den untersuchten Frameworks qualitativ bewertet, ob die im vorherigen Kapitel definierten Aufgaben alleine mithilfe der von dem Framework bereitgestellten Funktionalität gelöst werden können. Dem gleich stehend ist es, wenn der Framework-Ersteller eine Bibliothek anbietet, welche zwar ausgelagert ist und auch für sich alleinstehend verwendet werden kann, aber explizit dafür ausgelegt ist, mit dem entsprechenden REST-Framework genutzt zu werden. Dies drückt sich zum Beispiel darin aus, dass in der Dokumentation des Frameworks aktiv auf die Bibliothek verwiesen wird. Bietet ein Framework keine Möglichkeit, eine Aufgabe zu lösen, so wird diese mit der primitivsten alternativen Bibliothek gelöst. Ist es zum Beispiel nicht möglich, Daten aus einer Datenbank

abzurufen, wird eine Bibliothek verwendet, welche lediglich das Abrufen der Daten via SQL, also den primitivsten Weg, ermöglicht.

3.3.4 Wichtige Architekturprinzipien und besondere Sprachfunktionen

Steven Solleeder

Im Laufe der Zeit entwickelte sich in der Informatik die Hardware so stark weiter, dass immer weniger darauf geachtet werden musste, ob ein Programm bezogen auf seine Performance perfekt optimiert ist. Stattdessen konnten zunehmend mehr Architekturprinzipien in Anwendungen umgesetzt werden. Dies zeigt sich insbesondere darin, dass jede neue Programmiersprache von sich aus mehr Möglichkeiten bietet, verschiedenste Architekturprinzipien auf einfachere Weise umzusetzen. Eine gute Architektur hat viele Ziele. Ein sehr wichtiges davon ist, trotz einer großen Menge an Code und Funktionalität die Codebasis weiterhin leicht verstehen zu können. Ebenso von Bedeutung ist das Erweitern von Software um neue Features, ohne dabei bestehende Teile im Wesentlichen verändern zu müssen. Das führt auch dazu, dass die Testbarkeit erhöht wird, was ebenfalls ein wichtiges Ziel von einer guten Architektur ist. Genauso muss es auch möglich sein, Teile des Codes einfach austauschen zu können. Wollte man gute Architektur in einem Satz zusammenfassen, so könnte man sagen, dass es darum geht, Software so zu entwickeln, dass diese aus einzelnen abgeschlossenen Einheiten besteht oder noch kürzer: "Teile und herrsche".

Folglich soll in der vorliegenden Arbeit für die verschiedenen Frameworks untersucht werden, was die einzelnen Einheiten sind und ob diese das Single-Responsibility-Prinzip einhalten. Des Weiteren wird begutachtet, wie stark die Kopplung zwischen diesen Teilen ist und wie einfach neue Endpunkte und Ressourcen hinzugefügt werden können. Auch gilt es zu prüfen, inwieweit Middlewares zum Einsatz kommen und wie und an welcher Stelle diese einfügt werden können. Stets soll dabei betrachtet werden, ob Erweiterungen aller Art einen Einfluss auf bereits bestehende Teile nehmen. All diese Bewertungen finden qualitativ statt.

Ebenso beziehungsweise auch im Zusammenhang mit der Architektur soll die genutzte Sprache in einem kleinen Rahmen und im Kontext des jeweiligen Frameworks bewertet werden. Neben den Sprachfunktionen, welche sich so gut wie alle modernen Programmiersprachen teilen, z. B. das Erstellen von Klassen oder das Definieren von Prozeduren, besitzt jede

Programmiersprache eigene Sprachkonstrukte. Diese haben entweder den Zweck, häufig ähnlich vorkommenden Code wesentlich kürzer zu schreiben beziehungsweise Boilerplate-Code zu reduzieren oder aber sie dienen dazu, eine Intention des Softwareentwicklers auf direkterem Wege zu äußern, als wenn übliche Sprachfeatures genutzt werden würden. Dies soll qualitativ im Rahmen dieser Arbeit beurteilt werden.

3.3.5 Performance

Isabell Waas

Bei RESTful Webservices spielt die Performance wie so oft in der Informatik eine ganz zentrale Rolle. Wenn beispielsweise eine Webanwendung, welche von Nutzern bedient wird, auf eine API zurückgreift, ist eine angenehme und flüssige Interaktion sehr wichtig. Kommt es immer wieder zu langen Ladezeiten aufgrund einer langsamen Kommunikation mit der API, ist es sehr wahrscheinlich, dass der Besucher die Seite schnell wieder verlässt und sich nach einer Alternative umsieht. Besonders wichtig ist die Performance auch bei zeitkritischen Anwendungen wie zum Beispiel einem Live Ticker, welcher mittels Polling immer wieder bei einem Webservice anfragt, ob es Änderungen gibt. Je länger der Webservice in diesem Fall für die Bearbeitung einer Anfrage braucht, umso größer ist die Verzögerung, mit der die Anwendung die neuste Meldung anzeigen kann. Zuletzt kann ein Webservice bei einer guten Performance mit gleicher Leistung mehr Nutzer bedienen. Insgesamt können sich durch eine gute Performance eine Menge Kosten gespart werden.

Auch im Rahmen dieser Arbeit soll untersucht werden, mit welchem Framework der performanteste RESTful Webservice entwickelt werden kann. Hierzu sind sowohl gleichbleibende Testbedingungen als auch ein sinnvolles Testskript wichtig, da nur dann eine aussagekräftige Vergleichbarkeit gegeben ist.

Testbedingungen

Isabell Waas

Was die Testbedingungen betrifft, so soll darauf geachtet werden, dass die Ausgangssituation der Messung bei allen Webservices so identisch wie möglich ist. Aus diesem Grund wird stets derselbe Computer verwendet, sodass die zur Verfügung stehende Leistung immer gleich ist. Damit identische Temperaturumstände herrschen, wurde das Testskript, welches

später noch erläutert wird, unmittelbar vor der tatsächlichen Messung für jedes Framework einmal durchgeführt. Bei dem Computer handelt es sich um ein MacBook Pro 16 Zoll Basemodel, welches einen Apple M1 Pro Prozessor und 16GB RAM besitzt. Auf diesem ist das Betriebssystem macOS Ventura 13.2.1 installiert. Abgesehen von dem jeweils zu testenden Webservice werden auf dem Gerät nur die Hintergrundprozesse ausgeführt, welche von macOS selbst benötigt werden, wodurch die gesamte Leistung des Computers für den Webservice genutzt werden kann. Die für den jeweiligen Webservice benötigte Datenbank läuft darüber hinaus mit Docker¹⁴. Sowohl in den Datenbanken als auch in den Implementierungen wird festgelegt, dass 150 gleichzeitige Verbindungen zur Datenbank erlaubt sind.

Messtool Artillery

Isabell Waas

Als Werkzeug zum Messen der Performance wird auf das Open-Source Kommandozeilentool Artillery¹⁵ zurückgegriffen. Dieses ist sehr einfach zu nutzen, kann neben RESTful Webservices auch viele anderen Backend-Anwendungen testen und ist durch Plugins erweiterbar. Zudem sind mit Artillery zwei Arten von Performance-Tests möglich: Tests, welche die Belastbarkeit der Anwendung testen, und funktionale Tests, die prüfen, ob die Anwendung die erwarteten Ergebnisse liefert. Hierfür wird zunächst ein Testskript in der Auszeichnungssprache YAML geschrieben. In diesem können ein oder mehrere Scenarios definiert werden, welche die einzelnen auszuführenden Schritte beinhalten. Im Falle von Webservices sind das die aufzurufenden Endpunkte. Ein Scenario kann auch Schleifen, Bedingungen und anderen JavaScript-Code umfassen und so komplexe Testabläufe festlegen. Daneben werden in dem Testskript auch Phasen angegeben, die bestimmen, wie viele virtuelle Nutzer wie oft in welchem Zeitraum Anfragen an den Webservice stellen sollen. Das im Rahmen dieser Bewertung verwendete Testskript wird später vorgestellt und erläutert. Das Testskript wird letztendlich über die Kommandozeile ausgeführt. Anschließend kann mit dem Befehl „report“ ein Testbericht in Form einer HTML-Seite generiert werden [10]. Nachdem folglich die Performance des Szenarios, das alle Endpunkte beinhaltet, für die im Rahmen der Arbeit entstehenden Webservices getestet wurde, kann für jeden Webservice ein Report erstellt werden. Die Reports werden dann miteinander verglichen.

¹⁴<https://www.docker.com/>

¹⁵<https://www.artillery.io/>

Testskript

Steven Solleeder

Als Nächstes soll das für die Bewertung verfasste Testskript näher beleuchtet werden.

```

1 config:
2   phases:
3     - name: "Warm up"
4       duration: 30
5       arrivalRate: 10
6       rampTo: 30
7     - name: "Constant high load"
8       duration: 30
9       arrivalRate: 30
10      http:
11        timeout: 10
12 scenarios:
13   - flow:
14     - post:
15       url: "/v1/books"
16       json:
17         isbn: "978-3-5222-0260-2"
18         title: "Die endliche Geschichte"
19         author: "Michael Anfang"
20       capture:
21         - json: "$[id]"
22           as: "createdBookId"
23     - get:
24       url: "/v1/books/{{ createdBookId }}"
25     - patch:
26       url: "/v1/books/{{ createdBookId }}"
27       json:
28         title: "Die unendliche Geschichte"
29         author: "Michael Ende"
30     - delete:
31       url: "/v1/books/{{ createdBookId }}"
32     - loop:
33       - post:
```

```

34     url: "/v1/books"
35     json:
36         isbn: "978-3-5222-0260-2"
37         title: "{{ $randomString() }}"
38         author: "{{ $randomString() }}"
39     count: 10
40     - loop:
41         - get:
42             url: "/v1/books"
43             qs:
44                 page: "{{ $loopElement }}"
45                 per_page: 10
46             over:
47                 - 1
48                 - 2
49                 - 3
50                 - 4
51                 - 5
52                 - 6
53                 - 7
54                 - 8
55                 - 9
56                 - 10

```

Listing 4: Testskript zur Messung der Leistung

Das Testskript hat zwei wesentliche Ziele.

Einerseits soll ermittelt werden, welches Framework und welche Sprache am besten darin ist, die zur Verfügung stehende Leistung optimal zu nutzen. Dazu muss gewährleistet werden, dass jedes Framework die gleiche Anzahl an Anfragen auch wirklich beantwortet. Deswegen soll die Last so bestimmt sein, dass es zu keiner Überlastung kommt. Überlastungen würden sich vor allem in Timeouts oder - je nach Framework - in dem Statuscode 500 ausdrücken. Im Testskript wird dazu festgelegt, wie viele virtuelle Nutzer zu welchem Zeitpunkt anfangen, Anfragen zu stellen. Jeder virtuelle Nutzer stellt dabei alle Anfragen nacheinander, welche in sogenannten flow definiert sind. Konkret wird im Testskript vorgegeben, dass während den ersten 30 Sekunden in der ersten Sekunde 10 virtuelle Nutzer

ihre Anfragen beginnen, jedoch die Menge neuer virtueller Nutzer linear steigt, sodass in der dreißigsten Sekunde 30 weitere virtuelle Nutzer beginnen. In den folgenden 30 Sekunden starten jede Sekunde 30 neue virtuelle Nutzer mit ihren Anfragen.

Andererseits soll das Testskript umfassend sein. Das bedeutet, dass möglichst jeder definierte Endpunkt genutzt wird. Nur dann ist das Testskript realistisch, denn im produktiven Einsatz würde auch nicht nur einer oder wenige Endpunkte dauerhaft genutzt werden. Dazu wird festgelegt, dass ein virtueller Nutzer zuerst ein Buch erstellt, dieses anschließend abruft und daraufhin aktualisiert. Danach erstellt er noch 10 weitere Bücher und ruft zum Schluss via Pagination die ersten 10 Seiten mit jeweils 10 Büchern ab.

Zum Ausführen wird dabei folgendes CLI-Kommando verwendet.

```
1 artillery run --insecure --target http://localhost:12345 --output=result.json
config .yml
```

Listing 5: Kommando zum Durchführen des Artillery-Testskripts

Da die Server kein Zertifikat besitzen, sollte zum Unterdrücken von Warnungen das insecure-Flag verwendet werden. Da das Testskript für verschiedene Server und damit für verschiedene Adressen verwendet werden soll, wird diese auch über das CLI übergeben. Um das Ergebnis festzuhalten und in eine HTML-Datei zu konvertieren, wird es in eine JSON-Datei geschrieben. Zum Umwandeln des Outputs in eine HTML-Datei wird dabei folgender Befehl verwendet.

```
1 artillery report result.json
```

Listing 6: Kommando zum Umwandeln eines Artillery-Ergebnisses in eine HTML-Datei

4 Betrachtung und Bewertung der einzelnen Programmiersprachen und Frameworks

Isabell Waas

Damit wurden die Testbedingungen umfassend erläutert, insbesondere die umzusetzenden Aufgaben und die Kriterien für die Bewertung. Daher kann nun mit der Implementierung und Beurteilung der vier RESTful Webservices begonnen werden. Hierbei wird stets nur das Wesentliche beschrieben und erläutert, da viele der gezeigten Codeausschnitte größtenteils für sich selbst sprechen. Zudem wird sich bei dem Codestil stets an den jeweils geltenden Konventionen orientiert.

Die vollständigen Implementierungen, worin auch die jeweils verwendeten Versionen aller Technologien sowie die Startanweisungen jedes Projektes nachgelesen werden können, sind dieser Arbeit beigelegt.

4.1 C# mit ASP.NET Core

4.1.1 Einführung in die Sprache und das Framework

Steven Solleder



Abbildung 17: C# Logo (inoffiziell) [37]



Abbildung 18: .NET Core Logo (inoffiziell) [32]

Die objektorientierte Programmiersprache C#¹⁶, ausgesprochen „C Sharp“, wurde von den C-ähnlichen Sprachen, vor allem Java, stark inspiriert. Sie war ursprünglich dafür gedacht, ausschließlich unter dem Betriebssystem Windows in Verbindung mit dem zur Entwicklung und Ausführung benötigten .NET-Framework genutzt zu werden. Einer der Gründe hierfür ist, dass sowohl C# als auch Windows von dem Technologieunternehmen Microsoft ins Leben gerufen wurden. Jedoch wurde vor einigen Jahren .NET Core entwickelt, welches eine Art Modernisierung des .NET-Frameworks darstellt, die nicht mehr nur unter Windows funktioniert. Dadurch wurde es möglich, C# auf allen gängigen Plattformen, darunter Linux und macOS zu verwenden, wodurch die Sprache für mehr Personen nutzbar geworden ist. Mittlerweile wurden das .NET-Framework und .NET Core zu einem einzigen Projekt vereinigt, welches als .NET bezeichnet wird und C# gehört zu den beliebtesten Programmiersprachen weltweit (siehe Kapitel 3.1). Dies liegt unter anderem daran, dass mit C# erstellte Anwendungen durch Features wie Typsicherheit sicher und stabil sind. Alles in allem sind die Einsatzzwecke der 2001 veröffentlichten Sprache vielseitig. Neben dem Erstellen von Desktop-Anwendungen und mobilen Apps sowie der Web- und Backend-Entwicklung kann C# auch für Konsolenprogramme oder Videospiele genutzt werden [9, 21, 31].

In dieser Arbeit soll ein Webservice mit C# unter Einsatz des Open-Source Frameworks ASP.NET Core¹⁷ entwickelt werden. Hierbei soll angemerkt werden, dass ASP.NET Core, welches plattformübergreifend funktioniert, und ASP.NET, das nur unter Windows nutzbar

¹⁶<https://learn.microsoft.com/de-de/dotnet/csharp/>

¹⁷<https://learn.microsoft.com/de-de/aspnet/core/?view=aspnetcore-7.0>

ist, nebeneinander existieren und nicht wie das .NET-Framework und .NET Core zu einem Projekt vereint wurden [18].

4.1.2 Implementierung des RESTful Webservices

Steven Solleder

Als Erstes soll ein RESTful Webservice mit C# und ASP.NET Core entwickelt werden. Dieser soll die in Kapitel 3.2 festgelegten Aufgaben implementieren.

Definition von Ressourcen und Data Transfer Objects

Ein jeder RESTful Webservice verarbeitet in irgendeiner Weise Daten und benötigt dazu sowohl Model-Klassen als auch Data Transfer Objects. Model-Klassen spiegeln dabei die Struktur der Business-Logic der Anwendung wieder, während DTOs Datenformate speziell für die Kommunikation zwischen zwei Akteuren darstellen. Im Folgenden ist das DTO CreateBook zu sehen.

```
1 public record class CreateBook ([Required] [IsbnNumber] string isbn, [Required] [
    MinLength(1)] [MaxLength(300)] string title, [Required] [MinLength(1)] [
    MaxLength(100)] string author);
```

Listing 7: C#-Code des DTOs CreateBook

In dem Listing fällt auf, wie kurz die Deklaration der Klasse ist. Dies wird durch das von C# vor nicht allzu langer Zeit eingeführte Keyword „record“ möglich. Es sorgt dafür, dass alle für eine Model-Klasse üblichen Anforderungen automatisch zur Compile Time erstellt werden und nicht manuell geschrieben werden müssen. Zu diesen gehören insbesondere eine equals-Methode für den Vergleich auf Basis der Attributwerte statt der Referenz des Objektes, ein Konstruktor sowie die Getter- und Init-Methoden. Was die Init-Methoden betrifft, so sorgen sie dafür, dass die Daten nach der Objekterstellung nicht mehr geändert werden können, was zu einer erhöhten Datenkonsistenz im gesamten Programm führt. Es ist hierbei auch möglich, explizit anzugeben, dass keine Init-Methoden, sondern Setter-Methoden genutzt werden, wodurch die Eigenschaften des Objektes nach seiner Erzeugung veränderbar sind. Die im Listing genutzten Attribute wie zum Beispiel „[Required]“ werden erst später bei der Validierung benötigt und werden in diesem Zusammenhang näher erläutert.

Bei den Model-Klassen sind hingegen kaum C#-Attribute notwendig. Der folgende Codeausschnitt zeigt die Klasse Book, welche die Ressource definiert, für die anschließend Endpunkte entwickelt werden.

```

1 public record class Book(uint? id, string isbn, string title, string author) {
2     [Key] public uint? id { get; set; } = id;
3     public string isbn { get; set; } = isbn;
4     public string title { get; set; } = title;
5     public string author { get; set; } = author;
6 }
```

Listing 8: C#-Code der Model-Klasse Book

Da es bei dem später erläuterten PATCH-Endpunkt notwendig ist, dass ein Buch auch im Nachhinein verändert werden kann, wird explizit angegeben, dass die Attribute keine Init-Methoden, sondern Setter-Methoden erhalten sollen. Darüber hinaus muss nichts außer der Primärschlüssel in Form von „[Key]“ definiert werden, damit die Klasse als Entität gilt, die später in der Datenbank gesichert wird. Hierfür existiert ein ORM, welches im nächsten Abschnitt näher erklärt wird.

Abruf eines Objekts

Nachdem nun die Ressourcen und DTOs angelegt wurden, soll es ermöglicht werden, ein gespeichertes Buch auf Basis einer im Request gesendeten Id abzurufen. Zur Interaktion mit der Datenbank nutzt ASP.NET Core das ORM Entity Framework Core (EF Core)¹⁸. Dieses wird offiziell von ASP.NET Core empfohlen, ist explizit und konstant in der Dokumentation von ASP.NET Core zu finden und wurde genauso wie das Webframework selbst von Microsoft entwickelt. Für die Einrichtung des ORMs wird wenig Code benötigt. Dieser ist in dem folgenden Listing abgebildet.

```

1 public class BooksContext : DbContext {
2     public DbSet<Book> Books { get; init; } = null!;
3
4     public BooksContext(DbContextOptions<BooksContext> options) : base(options) {
```

¹⁸<https://learn.microsoft.com/de-de/ef/core/>

```

5     if (options == null) throw new ArgumentNullException(nameof(options));
6 }
7
8 protected override void OnModelCreating(ModelBuilder modelBuilder) {
9     modelBuilder.Entity<Book>().ToTable("Books");
10 }
11 }
```

Listing 9: C#-Code für den BooksContext

Zur Datenbankinteraktion muss eine Klasse geschrieben werden, die von DbContext erbt und öffentliche Properties des Typs DbSet<T> definiert, welche via Reflections von DbContext gesetzt werden. Diese DbSet-Properties stellen zahlreiche übliche Methoden zur Datenmanipulation zur Verfügung. DbContext bietet zudem einige Lifecycle-Methoden an, die überschrieben werden können. In dem beispielhaften Webservice in dieser Arbeit wird OnModelCreating genutzt, um zu definieren, welche Model-Klasse von EF Core berücksichtigt werden soll. Abgesehen davon muss über den Konstruktor die Konfiguration der Datenbank übergeben werden. Die Übergabe der Konfiguration geschieht schließlich bei der Registrierung des DbContexts im Startpunkt der Anwendung, wie der unten stehende Code zeigt.

```

1 builder.Services.AddDbContext<BooksContext>(dbContextOptions => {
2     dbContextOptions.UseMySql("Server=localhost;port=23495;Database=libraryapi;User
3         Id=libraryapi;Pwd=1234;Max Pool Size=150;", new MariaDbServerVersion(new
4             Version(10, 10, 3)),
5             mySqlOptions => {
6                 mySqlOptions.EnableStringComparisonTranslations();
7             });
8 });
9 }
```

Listing 10: C#-Code der Registrierung des DbContexts

Um nun Endpunkte, welche nach dem REST-Prinzip arbeiten, zu definieren, wird ein sogenannter Controller angelegt. Dieser ist eine Klasse, die von der Klasse ControllerBase erben muss. Mit dem Klassen-Attribut „ApiController“ werden dem Entwickler einige repetitive Aufgaben abgenommen.

```

1 [ApiController]
2 [Route( "/v1/books" )]
3 public class BooksController : ControllerBase {
4     private readonly BooksContext _booksContext;
5
6     public BooksController(BooksContext booksContext) => _booksContext =
7         booksContext;
8
9     [HttpGet( "{id}" )]
10    public async Task<ActionResult<Book>> GetBook([ Required ] uint id) {
11        Book? searchedBook = await _booksContext.Books.FindAsync(id);
12        if (searchedBook == null) {
13            return NotFound();
14        }
15        return searchedBook;
16    }
17    // Weitere Methoden
18 }
```

Listing 11: C#-Code des BooksControllers mit der Methode GetBook

Durch die Definition von Übergabeparametern im Konstruktor und die Registrierung deren Klassen als Service im Startpunkt des Programms (siehe Kapitel 4.1.2) werden schließlich automatisch passende Objekte übergeben. Dieses Prinzip heißt Dependency Injection und wird beispielsweise in dem Konstruktor des BooksControllers bei dem Übergabeparameter des Typs BooksContext angewandt (siehe Listing 11).

Jeder Endpunkt stellt zudem eine eigene Methode in der Klasse BooksController dar und die Attribute auf Klassen- und Methodensignaturebene legen die Routen fest. Da es sich bei C# um eine statisch typisierte Programmiersprache handelt, muss für jede Methode ein Rückgabetyp definiert sein. Um verschiedene Arten von Objekten zurückzugeben zu können, wird Polymorphismus genutzt. Dabei muss ein Typ zurückgegeben werden, welcher das Interface IActionResult implementiert. Dieses Kriterium erfüllen bereits zahlreiche Klassen, darunter beispielsweise die Klasse NotFoundResult. Um nicht „new NotFoundResult()“ schreiben zu müssen, bietet ControllerBase für häufig benötigte Rückgaben kürzere Methoden an, wie

z. B. `NotFound()`. Zusätzlich gibt es die Klasse `ActionResult<T>`. Diese stellt eine Standardimplementierung dar, welche es erlaubt, über den Generic `T` eine Klasse festzulegen, die dann mit dem Statuscode 200 direkt in der Methode zurückgegeben werden kann. Da in der Methode asynchrone Methodenaufrufe geschehen, wird dieser Rückgabetyp zusätzlich im Generic der Klasse `Task` gewrapped.

Der Endpunkt für diese Aufgabe wird in der Methode `GetBook` definiert (siehe Listing 11). Mithilfe einer einzigen Methode des ORMs kann ein Buch über seine Id gesucht werden. Variablen aus der Route, wie „`id`“, sind via Übergabeparameter nutzbar. Dabei können Einschränkungen getroffen werden. Näheres wird im Abschnitt zur Validierung erläutert.

Abruf aller Objekte mit Pagination, Sortierung und Filterung

Als Nächstes wird das Abrufen mehrerer Objekte betrachtet, die aufsteigend nach ihrer Id sortiert sind. Dabei soll es zudem die Möglichkeit geben, diese Objekte mit einem Suchbegriff zu filtern. Ebenso sollen die Objekte nur mit Pagination abgerufen werden können.

```

1 [HttpGet]
2 public async Task<BooksResponse> GetBooks([FromQuery] [Required] [Range(1, uint
3     MaxValue)] uint page, [FromQuery] [Required] [Range(1, 50)] uint per_page, [
4     FromQuery] [MinLength(1)] [MaxLength(30)] string? query) {
5     IQueryable<Book> queryable = from books in _booksContext.Books orderby books.id
6         ascending select books;
7
8     if (query != null) {
9         queryable=queryable.Where(book => book.id.ToString()!.Contains(query,
10             StringComparison.OrdinalIgnoreCase) ||
11                 book.isbn.Contains(query, StringComparison.OrdinalIgnoreCase) ||
12                 book.title.Contains(query, StringComparison.OrdinalIgnoreCase) ||
13                 book.author.Contains(query, StringComparison.OrdinalIgnoreCase));
14     }
15
16     return new BooksResponse(
17         ((uint) await queryable.CountAsync()) / per_page + 1,
18         await queryable
19             .Skip((int) ((page - 1) * per_page))
20             .Take((int) per_page)
21     );
22 }
```

```

17     .ToListAsync()
18 );
19 }
```

Listing 12: C#-Code der Controller-Methode GetBooks

Auffällig ist die Zeile 3 des Listings, in welcher eine SQL-ähnliche Syntax verwendet wird. Diese nennt sich Language Integrated Query (LINQ) und bietet zahlreiche Wege, auf einfachem Weg datenbankunabhängige Abfragen zu kreieren. Dabei wird stets ein Objekt zurückgegeben, welches das Interface `IQueryable` implementiert. Es ist anzumerken, dass LINQ-Anweisungen zur Compile Time überprüft werden. Mit der Zeile 3 ist die komplette Abfrage bis auf das Filtern bereits erledigt. Der Filter hingegen wird wieder über eine C#-übliche Syntax umgesetzt, da LINQ keine konditionalen Teile erlaubt. Auch hervorzuheben ist, dass `IQueryable` passende Methoden `Skip` und `Take` zur Pagination zur Verfügung stellt.

Erstellung eines Objekts

Auch das Erstellen eines Objektes soll beleuchtet werden. Hierzu bietet der bereits erwähnte Typ `DbSet` die Methode `Add` an. Nachdem Änderungen an der Datenbank durchgeführt wurden, muss stets `SaveChangesAsync` aufgerufen werden. Das erstellte Objekt bleibt dauerhaft in Verbindung mit dem `DbSet`, weshalb nach dem Aufruf von `SaveChangesAsync` einfach auf die von der Datenbank erstellte Id zugegriffen werden kann.

```

1 [HttpPost]
2 public async Task<IActionResult> CreateBook([FromBody] [Required] CreateBook
3     createBook) {
4     Book newBook = new Book(null, createBook.isbn, createBook.title, createBook.
5         author);
6     _booksContext.Books.Add(newBook);
7     await _booksContext.SaveChangesAsync();
8 }
```

Listing 13: C#-Code der Controller-Methode CreateBook

Änderung einzelner Attribute eines Objekts

Im Folgenden sollen einzelne Attribute eines Buches über einen PATCH-Endpunkt geupdatet werden können. Dessen Implementierung verläuft ohne spezielle Unterstützung des Frameworks. Zuerst wird das gewünschte Buch mithilfe des ORMs gesucht. Je nachdem, welche Daten der Aufrufende geschickt hat, wird das Buch entsprechend modifiziert. Die Änderungen werden dann in der Datenbank gespeichert.

```

1 [HttpPatch("{id}")]
2 public async Task<IActionResult> UpdateBook([Required] uint id, [FromBody]
3     UpdateBook updateBook) {
4
5     Book? searchedBook = await _booksContext.Books.FindAsync(id);
6
7     if (searchedBook == null) {
8         return NotFound();
9     }
10
11
12     if (updateBook.isbn != null) {
13         searchedBook.isbn = updateBook.isbn;
14     }
15
16     if (updateBook.title != null) {
17         searchedBook.title = updateBook.title;
18     }
19
20     await _booksContext.SaveChangesAsync();
21
22     return NoContent();
23 }
```

Listing 14: C#-Code der Controller-Methode UpdateBook

Löschen eines Objekts

Das Löschen eines Buches gehört ebenfalls zu den umzusetzenden Anforderungen. Hierbei muss wieder zuerst das zu entfernende Buch gesucht werden und, insofern es nicht gefunden wurde, eine Fehlermeldung zurückgegeben werden. Wurde es gefunden, wird eine Methode zum Löschen des Buches verwendet, die EF Core anbietet. Auch dieser Vorgang muss wieder persistiert werden.

```

1 [HttpDelete("{id}")]
2 public async Task<IActionResult> DeleteBook([Required] uint id) {
3     Book? searchedBook = await _booksContext.Books.FindAsync(id);
4     if (searchedBook == null) {
5         return NotFound();
6     }
7
8     _booksContext.Books.Remove(searchedBook);
9     await _booksContext.SaveChangesAsync();
10
11    return NoContent();
12 }
```

Listing 15: C#-Code der Controller-Methode DeleteBook

Validierung von erhaltenen Daten

Anschließend soll sich der Validierung gewidmet werden. Welche Pfadparameter oder Properties in Objekten aus dem Request-Body auf welche Weise validiert werden, wird durch vorangestellte Attribute festgelegt. Dabei werden von .NET schon einige Attribute für häufig benötigte Zwecke mitgeliefert, darunter zum Beispiel [Range(n, m)], welches festlegt, wie klein beziehungsweise groß eine Zahl minimal beziehungsweise maximal sein darf. Zu bemerken ist, dass diese Attribute beim Ankommen der Daten einmal geprüft werden. Danach ist es im Programm jederzeit möglich, die Objekteigenschaften zu ändern, insofern kein anderer Mechanismus der Sprache dies verhindert. Darüber können auch eigene Validierungs-Attribute definiert werden.

```

1  public class IsbnNumberAttribute : ValidationAttribute {
2      protected override ValidationResult IsValid(object? value, ValidationContext
3          validationContext) {
4          if (value == null) {
5              return ValidationResult.Success!;
6          }
7
8          if (value is not string potentialIsbn) {
9              return new ValidationResult("Value must be a string");
10         }
11
12         if (Regex.IsMatch(potentialIsbn, @"^(?=(:[^0-9]*[0-9])
13 {10}(:[^0-9]*[0-9])\{3\})$) [\d-]+$")) {
14             return ValidationResult.Success!;
15         }
16
17     }

```

Listing 16: C#-Code eines Attributs zur Validierung einer ISBN-Nummer

Wie das Listing zeigt, muss hierfür eine Klasse definiert werden, welche von ValidationAttribute erbt und die Methode IsValid überschreibt. Dabei wird über die Übergabeparameter insbesondere der zu überprüfende Wert übermittelt. Durch die Rückgabe, die von dem Typ ValidationResult ist, wird bestimmt, ob die Validierung erfolgreich war.

Behandlung von Fehlern

Zur Fehlerbehandlung kann gesagt werden, dass das bewusste Schicken von Fehlern an den Clienten über die Rückgabe eines entsprechenden Objekts realisiert wird, welches das Interface IActionResult implementiert. So erstellt zum Beispiel die Methode NotFound() ein NotFoundResult-Objekt. Da nicht für alle Fehler eine eigene Methode oder Klasse existiert, gibt es auch allgemeinere Wege zur Fehlerbehandlung wie zum Beispiel die Methode StatusCode, die ein Objekt des Typs StatusCodeResult erstellt. Der Methode StatusCode kann dabei jede beliebige Zahl als Statuscode sowie ein Objekt für den Body übergeben werden.

Für den Fall, dass ein Fehler, also eine Exception, geworfen wurde, ist es auch möglich, einen ExceptionHandler in der Datei Program.cs zu definieren.

```

1 app.UseExceptionHandler(errorApp => {
2     errorApp.Run(async (HttpContext context) => {
3         Exception? exception = context.Features.Get<IExceptionHandlerFeature>()?.Value;
4         context.Response.StatusCode = (int) HttpStatusCode.InternalServerError;
5     });
6 })

```

Listing 17: C#-Code eines ExceptionHandlers

Immer wenn eine Exception nicht explizit behandelt wurde, wird die Callback-Funktion in dem Listing aufgerufen. Über das HttpContext-Objekt kann dabei auf die geworfene Exception zugegriffen werden und gegebenenfalls auf besondere Weise reagiert werden.

Konfiguration des Servers

Selbstverständlich bedarf es auch der Konfiguration des Servers. Diese verteilt sich in dem Webservice mit ASP.NET Core auf drei Teile. Zum einen gibt es die Datei launchSettings.json, welche Informationen dazu enthält, wie eine Anwendung gestartet werden soll. Folglich wird diese Datei auch nur von der genutzten Integrated Development Environment (IDE), z. B. Visual Studio oder Jetbrains RIDER, betrachtet. In ihr steht zum Beispiel, unter welcher URL der Service zu finden ist und ob durch die IDE ein Browser mit der URL gestartet werden soll.

```

1 {
2     "$schema": "https://json.schemastore.org/launchsettings.json",
3     "profiles": {
4         "normal": {
5             "commandName": "Project",
6             "dotnetRunMessages": true,
7             "launchBrowser": false,
8             "applicationUrl": "http://localhost:37364",
9             "environmentVariables": {
10                 "ASPNETCORE_ENVIRONMENT": "Development"
11             }
12         }
13     }
14 }

```

```

11     }
12   }
13 }
14 }
```

Listing 18: Datei launchSettings.json des .NET-Projekts

Die zweite Datei zur Serverkonfiguration stellt appsettings.json dar. Diese enthält alle Konstanten, welche zur Laufzeit selbst von Bedeutung sind. Dabei gibt es sowohl vordefinierter Attribute, z. B. „AllowedHosts“ als auch die Möglichkeit, eigene Attribute zu definieren. Sollen für bestimmte Umgebungen abweichende Attributwerte gesetzt werden, so muss eine weitere Datei erstellt werden, bei der der Name der Umgebung nach dem ersten Punkt von „appsettings.json“ folgt. Für die Entwicklungsumgebung würde diese z. B. appsettings.development.json heißen.

```

1 {
2   "Logging": {
3     "LogLevel": {
4       "Default": "Information",
5       "Microsoft.AspNetCore": "Warning"
6     }
7   },
8   "AllowedHosts": "*"
9 }
```

Listing 19: Datei appsettings.json des .NET-Projekts

Ein weiterer Teil der Konfiguration findet in der Datei Program.cs statt, welche gleichzeitig der Startpunkt der Anwendung ist. Dort wird mithilfe des Builder-Patterns und unter Zuhilfenahme der Datei appsettings.json das Programm konfiguriert und gestartet. Da die Datei Program.cs C#-Code enthält, können hier dynamische beziehungsweise komplexe Konfigurationen erstellt werden. Es ist nicht zwingend notwendig, auf Konstanten aus der Datei appsettings.json zuzugreifen.

```

1 WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
2 builder.Services.AddControllers();
3 builder.Services.AddDbContext<BooksContext>(dbContextOptions => {
4     dbContextOptions.UseMySQL("Server=localhost;port=23495;Database=libraryapi;User
5         Id=libraryapi;Pwd=1234;Max Pool Size=150;", new MariaDbServerVersion(new
6             Version(10, 10, 3)),
7     mySqlOptions => {
8         mySqlOptions.EnableStringComparisonTranslations();
9     }
10 });
11 builder.Services.Configure<ApiBehaviorOptions>(options => {
12     options.InvalidModelStateResponseFactory = actionContext => {
13         List<string> errors = actionContext.ModelState.Values
14             .SelectMany(entry => entry.Errors)
15             .Select(error => error.ErrorMessage)
16             .ToList();
17         actionContext.HttpContext.Response.ContentType = "application/json";
18
19         return new BadRequestObjectResult(new ErrorsResponse(errors));
20     };
21 });
22 WebApplication app = builder.Build();
23 app.UseExceptionHandler(errorApp => {
24     errorApp.Run(async (HttpContext context) => {
25         Exception? exception = context.Features.Get<IExceptionHandlerFeature>() ??
26             Error;
27         context.Response.StatusCode = (int) HttpStatusCode.InternalServerError;
28     });
29 });
30 app.UseHttpsRedirection();
31 app.UseAuthorization();
32 app.MapControllers();
33 app.Run();

```

Listing 20: C#-Code der Datei Program.cs

Außerdem gibt es noch eine Projektdatei mit der Endung csproj, welche Information über das Projekt, darunter die verwendete .NET-Version, Einstellungen zur Sprache, den Projektnamen oder die verwendeten Abhängigkeiten, enthält.

4.1.3 Bewertung

Steven Solleeder

Somit ist die Entwicklung des Webservices mit C# und ASP.NET Core abgeschlossen. Es kann nun mit der Bewertung anhand der beschriebenen Kriterien (siehe Kapitel 3.3) begonnen werden.

Dokumentation und Wartung

Bevor mit der Programmierung einer Anwendung mit einer Programmiersprache und einem Framework begonnen wird, muss das Entwicklerteam über diese zumindest im Wesentlichen Bescheid wissen. Dabei dient die Dokumentation als erster Anlaufpunkt. Microsoft hat viele größere Entwicklungsthemen in seiner Obhut, zu denen auch .NET, inklusive C#, und ASP.NET Core gehören.

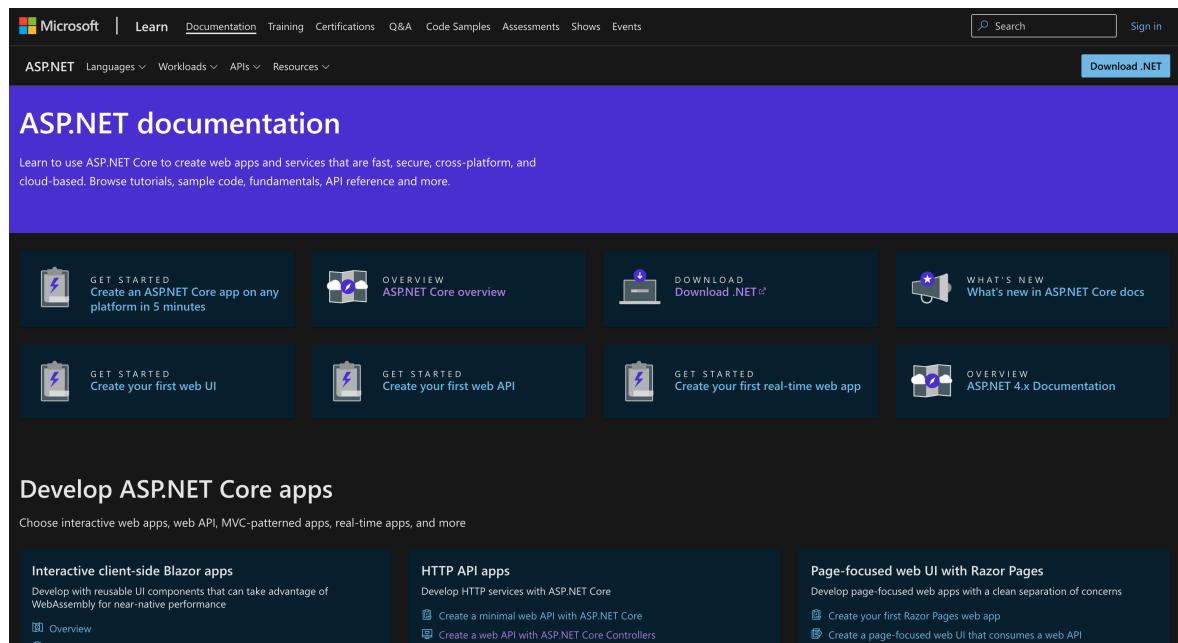


Abbildung 19: Seite zu einem der größeren Entwicklungsthemen ASP.NET Core von Microsoft [19]

Jedes größere Entwicklungsthema von Microsoft besitzt eine eigene Übersichtsseite in der Dokumentation. Dort gibt es oben spezifische Reiter und zentral einige Kästchen mit Unterabschnitten. Jeder Reiter und jedes Kästchen stellt ein eigenes relevantes Thema dar. Die Reiter sind bei den verschiedenen Entwicklungsthemen in etwa immer die gleichen. Die Kästchen enthalten verschiedene, häufig gesuchte Themen. Dabei wird auch Acht gegeben, dass die Themen sich sowohl an Einsteiger als auch an Fortgeschrittene richten. Diese Einfachheit der Darstellung und Vielfalt der Themen, welche sich an verschiedene Zielgruppen richtet, ist sehr positiv zu bewerten. Die Links in der Dokumentation führen oft zu einem Artikel einer Artikelsammlung, manchmal jedoch leider zu Artikeln eines ganz anderen größeren Entwicklungsthemas und seltene Male zu externen Webseiten wie zum Beispiel einer Playlist in YouTube¹⁹. Insofern es sich nicht um externe Seiten handelt, bleiben die Reiter und der Name des größeren Entwicklungsthemas auf der linken Seite gleich, jedoch kann der Anwender dennoch leicht die Übersicht verlieren, wenn er diese doch komplexere Struktur der Dokumentation nicht verstanden hat.

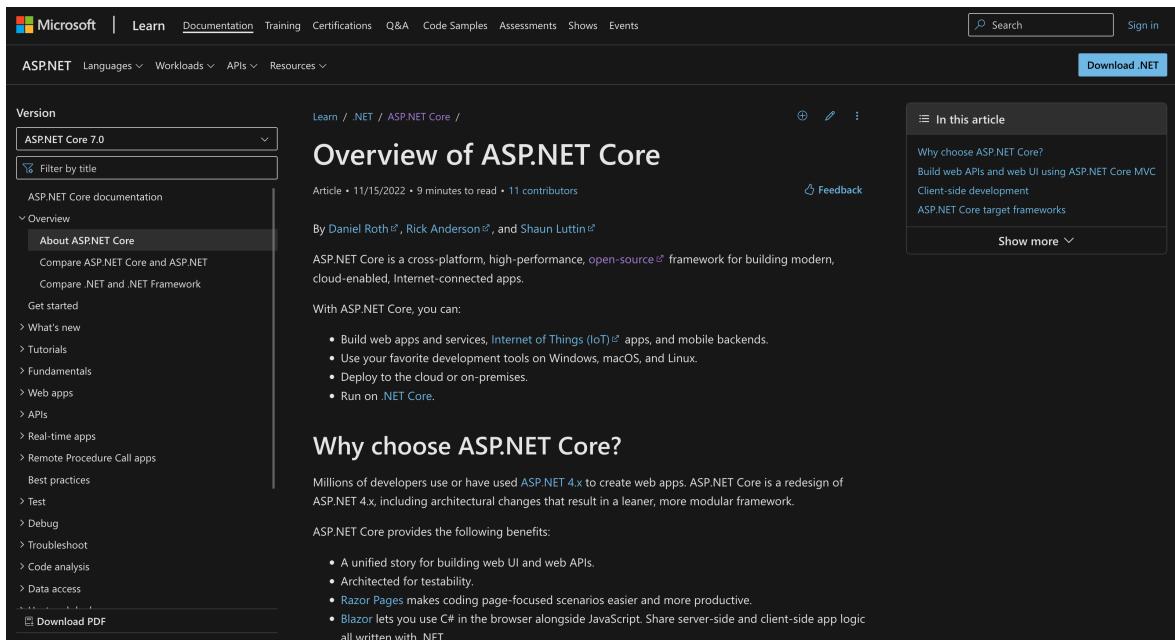


Abbildung 20: Ein Artikel der Artikelsammlung von ASP.NET Core [5]

Eine Artikelsammlung, die als eine Art abgeschlossenes Buch gesehen werden kann, besitzt auf der linken Seite das vollständige Inhaltsverzeichnis, welches sinnvoll gegliedert

¹⁹<https://www.youtube.com/>

ist. Auch ist es möglich, zwischen den verschiedenen Versionen der Artikelsammlung zu wechseln. Die eigentlichen Artikel werden dabei rechts vom Inhaltsverzeichnis dargestellt. Sie haben oben rechts eine Übersicht ihres Inhalts durch verlinkte Kapitelüberschriften, um gleich zu Themen springen zu können. Zudem sind sie meist sehr ausführlich und haben zahlreiche Codebeispiele. Am Ende verweisen sie auf andere interessante oder verwandte Themen. Die Links können auch zu Seiten außerhalb der gerade betrachteten Artikelsammlung führen, was zu einem ähnlichen Problem wie schon bei der Übersichtsseite führen kann. Beispiele für Artikelsammlungen sind „C#“, „.NET MAUI“, was ein GUI-Framework von .NET ist, oder „LINQ“. Erfreulicherweise gibt es auch für jedes Package von .NET Artikelsammlungen, wobei jeder Artikel eine Klasse und deren Methoden samt Beschreibungen darstellt. Leider gibt es dabei nicht immer Beschreibungen. Hier ist es sogar möglich, bei den Codebeispielen zwischen den verschiedenen .NET-Sprachen wie C# oder F# zu wechseln.

Generell hat die Dokumentation zu sehr vielen Themen Informationen zu bieten, was eventuell auch ihre etwas komplexere Struktur zumindest in Teilen rechtfertigt. Abgesehen davon finden sich noch verteilt auf anderen Seiten, die über die obere Navigationsleiste erreichbar sind, Codebeispiele oder Kurse, die sich jedoch stärker auf Microsoft Services wie Azure konzentrieren und die Nutzung von ASP.NET Core oder C# nicht im Speziellen behandeln.

Unabhängig von den gerade erläuterten Aspekten sticht ein Punkt eher negativ hervor: Die Dokumentation stellt die verwendete Sprache nicht selten beim Seitenaufruf auf die lokal verwendete Sprache um. Das ist problematisch, weil diese Übersetzung rein maschinell erfolgt und dies bei manchen Formulierungen und Wörtern zu Verwirrungen führen kann. Außerdem ist es den meisten Entwicklern vermutlich aufgrund der großen Menge an englischsprachigen Fachbegriffen generell lieber, die Dokumentation auf Englisch zu lesen. Zur Behebung dieses Problems gibt es sogar von anderen entwickelte Plugins.

Schließlich soll auf die Wartung des Frameworks eingegangen werden. Da ASP.NET Core in .NET enthalten ist, muss zur Bewertung der Wartung der Releasecycle von .NET betrachtet werden. Jedes Jahr im November erscheint eine Major-Version mit zahlreichen neuen Funktionen. Dabei ist jede ungerade Version, also zum Beispiel .NET 7 oder 9, eine Long Term Support Version (LTS-Version). LTS-Versionen werden drei Jahre nach dem Release gepflegt. Durch das Anbieten von LTS-Versionen muss weniger Zeit in Migrationen ausge-

geben werden, was zu niedrigeren Kosten führt. Die übrigen Versionen sind Standard Term Support-Versionen (STS-Versionen). Diese werden bis zum nächsten Release und sechs Monate darüber hinaus unterstützt. Patches erfolgen dabei in der Regel sowohl für LTS- als auch für STS-Versionen monatlich. Die jährlichen Major-Versionen, die regelmäßigen Patches, aber auch das Anbieten von LTS-Versionen sind als sehr positiv zu bewerten [17].

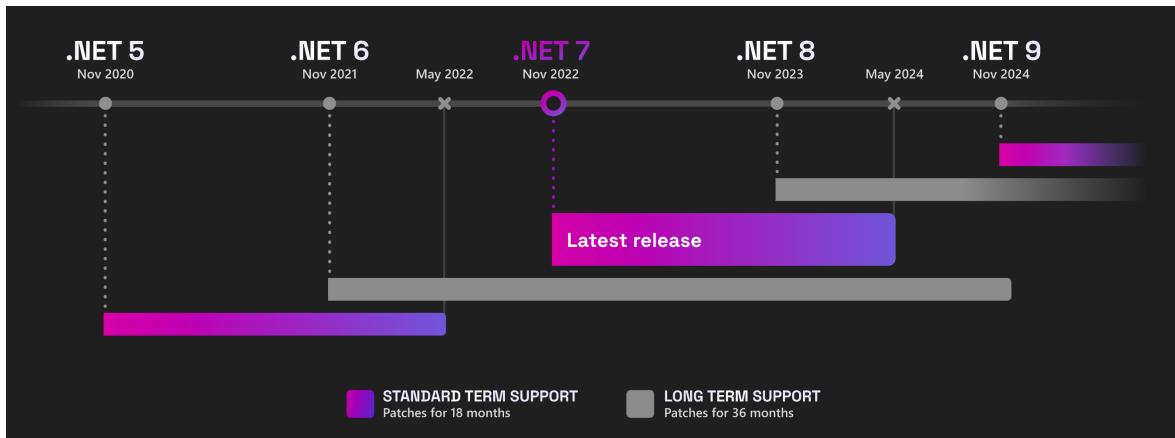


Abbildung 21: Releases von .NET [17]

Lines of Code

Nun sollen die Lines of Code aller Dateien, die C#-Code enthalten, mit dem Tool cloc ermittelt werden. Der Ordner Migrations wird dabei ausgenommen, weil er Dateien für die Migration der Datenbank beinhaltet, welche durch EF Core erstellt werden. Da es sich bei diesen nur um generierte Dateien handelt, sollen sie nicht von cloc erfasst werden. Insgesamt ergibt sich daher folgende Statistik für die Analyse der Lines of Code.

Language	files	blank	comment	code
C#	11	47	0	161

Tabelle 3: Ausgabe des Programms cloc bei der Implementierung mit C# und ASP.NET Core

Es muss erwähnt werden, dass die Bewertung der Lines of Code erst in Kapitel 5.1 erfolgen kann, da nur im Vergleich mit den Werten der anderen Frameworks erkannt werden kann, welche Anzahl an Codezeilen gut oder schlecht zu bewerten ist.

Notwendigkeit zusätzlicher Bibliotheken

Nachfolgend wird betrachtet, inwiefern zusätzliche Bibliotheken für den RESTful WebService benötigt werden. Diese Implementierung der Library API basiert selbstverständlich zum einen auf .NET und seinen mitgelieferten Klassen, zum anderen verwendet sie auch ASP.NET Core sowie das ORM Entity Framework Core. Dabei bringt ASP.NET Core alles bis auf den Datenbanktreiber mit sich. EF Core ist zwar eine eigene Bibliothek, welche auch autark in anderen Projekten verwendet werden kann, jedoch ist klar, dass EF Core primär für die Verwendung mit ASP.NET Core gemacht ist. Einerseits ist dies daran ersichtlich, dass in der Dokumentation und einigen Beispielen, insbesondere in denen über das Persistieren von Daten gesprochen wird, aktiv EF Core genutzt wird und auch auf diese Bibliothek verwiesen wird. Andererseits ist es bemerkbar, da EF Core aktiv mit ASP.NET Core zusammenarbeitet. So bietet das ORM eine eigene Extensionmethode für das Interface `IServiceCollection`, welches vom `WebApplicationBuilder` implementiert wird. Diese Extensionmethode erleichtert das Hinzufügen von `DbContext`-Klassen als Service, ohne dass eine weitere Konfiguration benötigt wird.

```

1 WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
2 builder.Services.AddControllers();
3 builder.Services.AddDbContext<BooksContext>(dbContextOptions => {
4     dbContextOptions.UseMySql("Server=localhost;port=23495;Database=libraryapi;User
5         Id=libraryapi;Pwd=1234;Max Pool Size=150;", new MariaDbServerVersion(new
6             Version(10, 10, 3)),
7             mySqlOptions => {
8                 mySqlOptions.EnableStringComparisonTranslations();
9             });
10 });

```

Listing 21: Nutzen der Methode `AddDbContext` in der Datei `Program.cs`

Wichtige Architekturprinzipien und besondere Sprachfunktionen

Als Nächstes sollen die Architektur sowie besondere Sprachfeatures beleuchtet werden. Im Folgenden ist der grobe Aufbau der Library API mit ASP.NET Core zu sehen, wobei nur Ordner, welche C#-Dateien enthalten sowie die bedeutendsten Dateien abgebildet wurden.

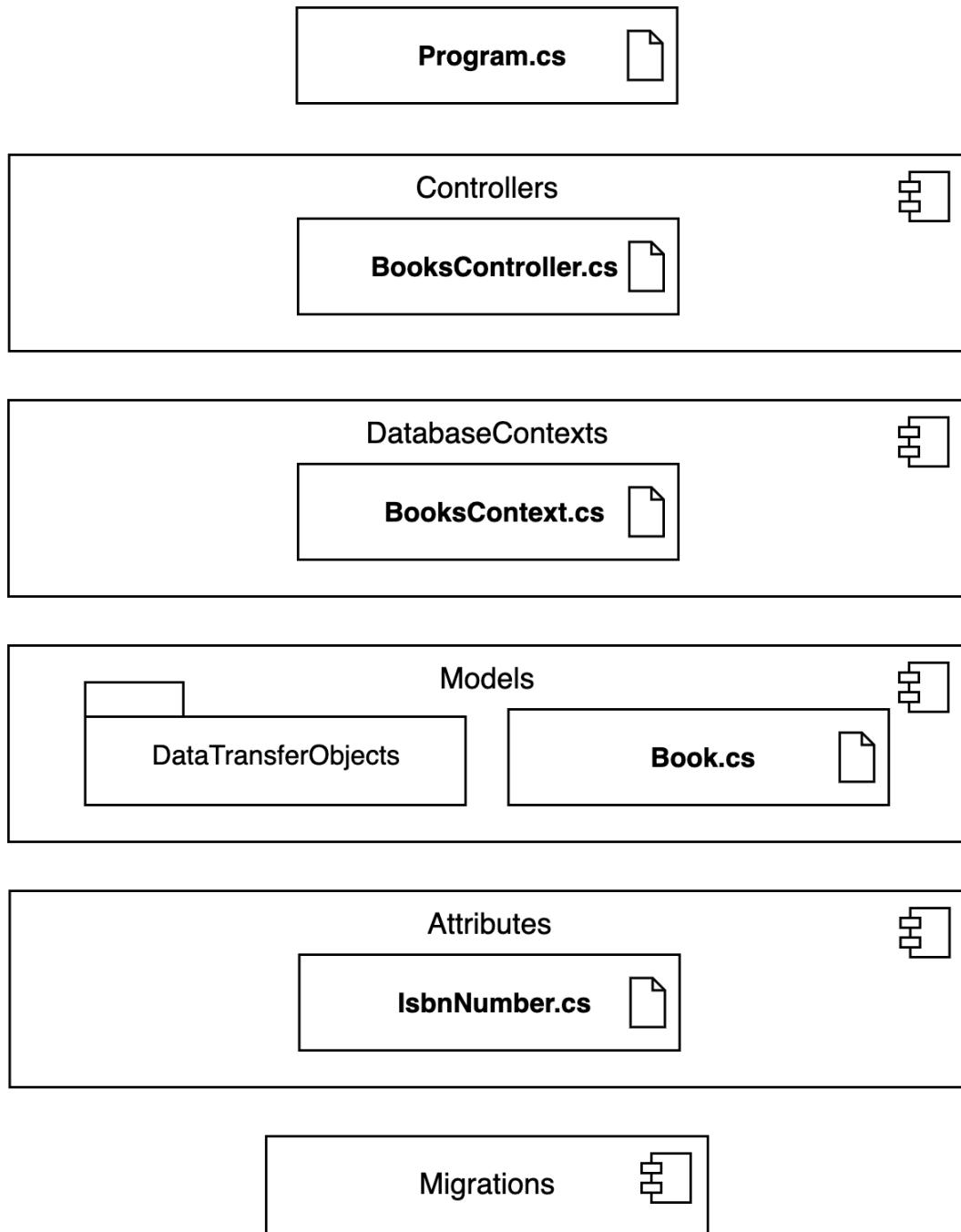


Abbildung 22: Architektur des Webservices mit ASP.NET Core

Dabei werden das Single-Responsibility-Prinzip und die Kapselung für so gut wie alle typischen Merkmale sowohl durch das Framework als auch durch Sprachfeatures sehr gut eingehalten. Diese typischen Merkmale sowie wie sie durch Framework und Sprache die zu untersuchenden Architekturprinzipien einhalten, soll im Folgenden erläutert werden.

Zum Hinzufügen der Datenbankanbindung muss eine vollständig eigene unabhängige Einheit, nämlich eine Klasse, welche von DbContext erbt, geschrieben werden. Dabei wird durch das Definieren von DbSet-Properties innerhalb der DbContext-Kindklasse festgelegt, welche Model-Klassen von dieser Kindklasse verwaltet werden. Dem Entwickler ist es selbst überlassen, wie viele DbContext-Klassen er schreibt und welche DbSet-Properties diese verwalten. Dies scheint ein guter Mittelweg zwischen architektonischer Vorgabe und Freiheit des Entwicklers zu sein.

Die Model-Klassen sind ebenfalls alle für sich eigenständige Klassen. Dabei wird dem Entwickler eine Menge Arbeit abgenommen, da C# in neueren Versionen die record-Klassen einführt, welche zahlreiche repetitive Aufgaben abnehmen, wie schon im Verlauf dieser Arbeit erläutert wurde.

Die Controller stellen ebenso eigene Klassen dar, wobei diese von der Klasse ControllerBase erben müssen. Ein Controller steht dabei immer für den ersten Teil eines Pfades, z. B. „/v1/books“, seine Methoden wiederum für die zweite Hälfte des Pfades, z. B. „/id“, und bestimmte HTTP-Verben. Welcher Pfad und welches HTTP-Verb genutzt wird, wird mithilfe von Attributen definiert. Durch das Schreiben einer zusätzlichen Methode oder eines neuen Controllers kann ganz unabhängig von den anderen Methoden eine neue Route erstellt werden. Alle Controller-Klassen werden dabei nur durch einen einzigen Methoden-Aufruf in der Datei Program.cs registriert und sind von diesem Moment an nutzbar.

Auch jedes Validierungs-Attribut stellt eine eigene Klasse dar, welche von der Klasse ValidationAttribute erbt sowie durch das Überschreiben der Methode IsValid beschreibt, wie ein Wert geprüft werden soll. Durch die Attribut-Funktionalität von C# ist es anschließend sehr einfach möglich, an verschiedensten Stellen Werte zu validieren.

Eigens geschriebene Klassen, die hauptsächlich dazu dienen, Funktionalität zur Verfügung zu stellen, lassen sich dank der integrierten Dependency Injection einfach an verschiedenen

Stellen nutzen, ohne dass sich der Entwickler um das Zusammenstecken des Objektgraphs kümmern muss. Als Erstes muss eine Klasse geschrieben werden, die ein bestimmtes Interface implementiert oder von einer bestimmten Klasse erbt. Danach muss diese registriert werden. Nun kann ein Objekt dieser Klasse über den Konstruktor übergeben und verwendet werden. Natürlich können darüber hinaus noch einige Feinheiten spezifiziert werden, z. B., ob immer das gleiche oder ein neues Objekt injiziert werden soll.

Zusammenfassend lässt sich sagen, dass ASP.NET Core versucht, jeden wesentlichen Teil in Klassen zu kapseln oder vom Entwickler kapseln zu lassen. Die Erstellung und Nutzung dieser Klassen wird dabei durch einige Features von ASP.NET Core beziehungsweise C# sehr unterstützt und gefördert, wie gerade in den verschiedenen Absätzen erläutert wurde. Diese starke Kapselung unterstützt das Testen, Erweitern und Austauschen der Teile enorm. Auch macht es die Anwendung leichter verständlich.

Performance

Weiterhin soll nun ein Blick auf die Performance des mit C# und ASP.NET Core entwickelten Webservice geworfen werden. Dies geschieht wie zuvor in Kapitel 3.3.5 angekündigt mit dem Kommandozeilen-Tool Artillery sowie dem ebenfalls bereits erläuterten, für diesen Zweck selbst verfassten Testskript. Nach der Testdurchführung wurde ein Report in Form einer HTML-Datei generiert, aus dem insbesondere die folgenden zwei Grafiken relevant sind.

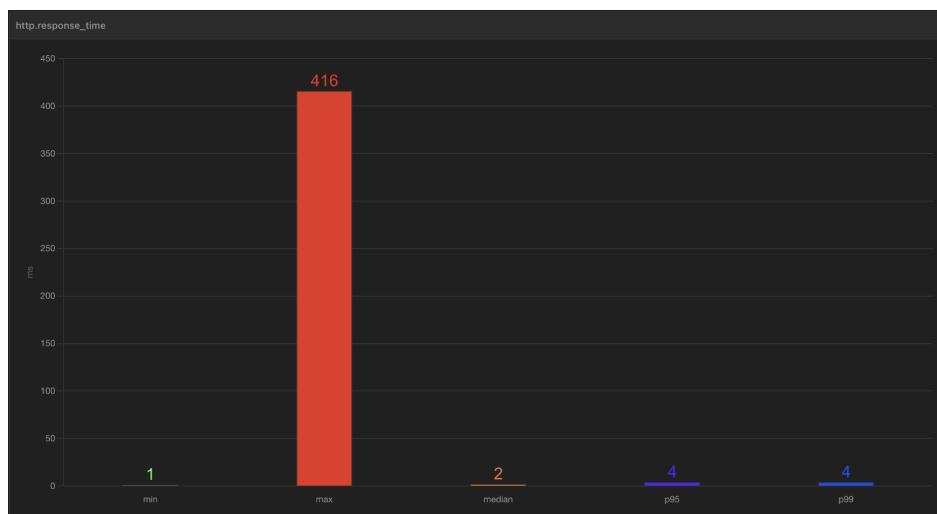


Abbildung 23: Ausgewählte Antwortzeiten für C# mit ASP.NET Core (Balkenwerte manuell hinzugefügt)

Das erste Diagramm visualisiert ausgewählte Antwortzeiten des Services. Die erste Säule spiegelt die Antwortzeit der Anfrage wider, die von allen getätigten Anfragen am schnellsten beantwortet wurde. Analog dazu stellt die zweite Säule dies für die maximale Antwortdauer dar. Die mittlere Säule „median“, welche auch „p50“ genannt werden könnte, zeigt an, welche Zeit 50% der Anfragen maximal gebraucht haben, um beantwortet zu werden. Entsprechend stellen die Säulen „p95“ und „p99“ diese Maximaldauer für 95% beziehungsweise 99% der Anfragen dar.

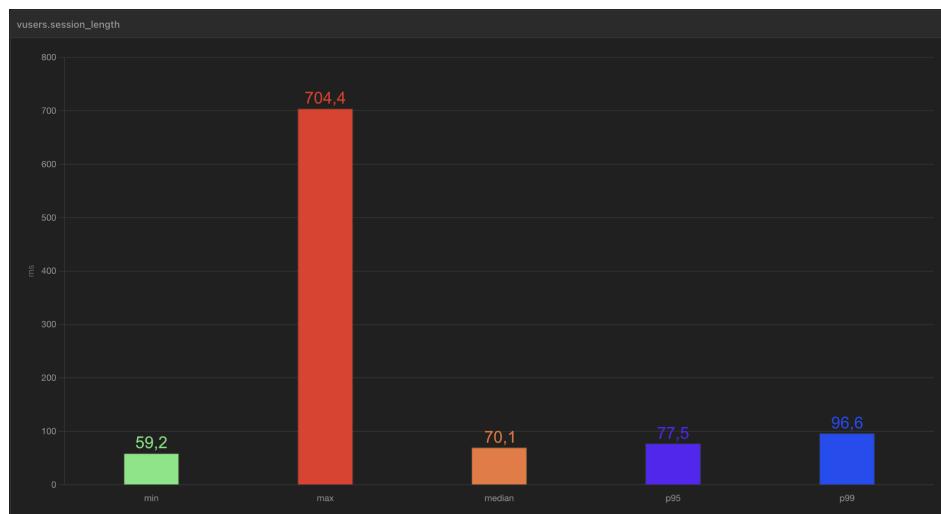


Abbildung 24: Ausgewählte Sessionlängen für C# mit ASP.NET Core (Balkenwerte manuell hinzugefügt)

Was die zweite Abbildung betrifft, so nutzt sie dieselben Säulenarten wie die vorherige, um ausgewählte Sessionlängen darzustellen. Eine Sessionlänge ist dabei wie bereits erklärt die Dauer für die Abarbeitung des Flows im Testskript durch einen virtuellen Nutzer.

Auch die Performance kann wie das Kriterium Lines of Code erst in Kapitel 5.1 bewertet werden, da hierfür der Vergleich aller Frameworks erforderlich ist.

4.2 Java mit Spring Boot

4.2.1 Einführung in die Sprache und das Framework

Isabell Waas



Abbildung 25: Java Logo [34]



Abbildung 26: Spring Logo [35]

Java²⁰ zählt bereits seit ihrer Veröffentlichung vor etwa 28 Jahren zu den meist genutzten Programmiersprachen weltweit. Nicht nur ist sie bis heute für viele Entwickler die erste Programmiersprache, die sie lernen, sie wird auch täglich in zahlreichen Unternehmen eingesetzt. Java wurde 1995 von Sun Microsystems herausgegeben, 2010 wurde Sun Microsystems jedoch von Oracle gekauft. Sie gehört zu den objektorientierten Sprachen und insbesondere ihre Syntax basiert auf den deutlich älteren Sprachen C und C++. Verglichen mit diesen beiden Sprachen, die zu Javas Anfangszeiten sehr beliebt waren, ist die Entwicklung mit Java jedoch wesentlich einfacher und weniger fehleranfällig. Grund hierfür ist vor allem, dass anspruchsvolle Funktionen wie Pointer in Java nicht vorhanden sind. Abgesehen von ihrer Einfachheit zeichnet sich Java durch ihre Portabilität aus, die häufig durch das Feature „write once, run anywhere“ beschrieben wird. Dieses sorgt dafür, dass ein unter Linux in Java verfasstes und kompiliertes Programm auf einer beliebigen Plattform, also beispielsweise einem Gerät mit Windows oder macOS, ausgeführt werden kann. Dies ist möglich, da Java beim Kompilieren in Bytecode umgewandelt wird, der auf jedem Gerät läuft, das die Java Virtual Machine (JVM) unterstützt. Letztere ist Teil der Java Runtime Environment (JRE). Die genannten Vorteile machen Java zu einer Allzweck-Programmiersprache, die für zahlreiche Programme eingesetzt werden kann – von serverseitigen Web- und Desktopanwendungen über Smartphone-Apps bis hin zu Videospielen und Applikationen für Haushaltsgeräte [47].

In dieser Arbeit soll Java zur Erstellung eines RESTful-Webservices genutzt werden. Da die Entwicklung trotz der Einfachheit der Sprache noch immer komplex ist, wird das seit

²⁰<https://docs.oracle.com/en/java/>

2003 existierende Open-Source Framework Spring²¹ von VMware angewandt, welches das älteste der betrachteten Frameworks ist. Spring besteht aus zahlreichen Modulen und für jede Anwendung kann flexibel entschieden werden, welche verwendet werden sollen. Eines dieser Module ist Spring Boot. Tatsächlich handelt es sich dabei eher um eine Erweiterung von Spring, welche das Erstellen und insbesondere Konfigurieren einer Spring-Anwendung automatisiert und damit stark vereinfacht [22]. Daher soll Spring Boot genutzt werden. Zum Generieren der Projektstruktur wird das Tool Spring Initializr²² genutzt, das in vielen IDEs integriert ist.

4.2.2 Implementierung des RESTful Webservices

Isabell Waas

Anschließend soll nun der RESTful Webservice mit der zweiten Kombination aus Sprache und Framework, Java und Spring Boot, entwickelt werden.

Definition von Ressourcen und Data Transfer Objects

Die Entwicklung eines RESTful Webservices beginnt stets mit dem Erstellen einer oder mehrerer Ressourcen. Das folgende Listing zeigt daher die Ressource „Book“ als Java Klasse, wobei die Getter und Setter für die verschiedenen Attribute zur besseren Übersichtlichkeit nicht mit abgebildet wurden, da diese keine Besonderheiten aufweisen.

```

1  @Entity
2  @Table(name = "books")
3  public class Book {
4      @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
5      private int id;
6
7      @Column(name="isbn", nullable=false)
8      @Pattern(regexp="^(978-?|979-?) ?\\d{1,5}-?\\d{1,7}-?\\d{1,6}-?\\d{1,3}$",
9          message="isbn is not in the correct format.")
10     private String isbn;
11
12     @Column(name="title", nullable=false)

```

²¹<https://spring.io/>

²²<https://start.spring.io/>

```

12  @Size(min=1, max=300, message="title must have between {min} and {max}
13    characters.")
14  @NotNull(message="title must not be empty.")
15  private String title;
16
17  @Column(name="author", nullable=false)
18  @Size(min=1, max=100, message="author must have between {min} and {max}
19    characters.")
20  @NotNull(message="author must not be empty.")
21  private String author;
22
23  protected Book() {}
24
25  public Book(String isbn, String title, String author) {
26    this.isbn=isbn;
27    this.title=title;
28    this.author=author;
29  }
// Getter und Setter
}

```

Listing 22: Java-Code der Ressource Book

Es fällt auf, dass bei der Definition der Ressource zahlreiche Annotations genutzt werden, die nützliche Informationen über das Programm bereitstellen.

Die Mehrheit der Annotations aus dem Codebeispiel gehören zu der Jakarta Persistence API (JPA), früher Java Persistence API genannt, welche eine Spezifikation zum Persistieren von Daten zwischen Objekten und einer relationalen Datenbank in Java ist. In dem Webservice wird das Modul Spring Data JPA²³ verwendet, das das Open-Source Tool Hibernate²⁴ nutzt. Dieses ist eine Implementierung der JPA-Spezifikation und beinhaltet unter anderem ein ORM [29]. Da die Bücher im zu entwickelnden Webservice in einer relationalen Datenbank gespeichert werden sollen, wird über die in dem Listing verwendeten JPA Annotations angegeben, dass die Ressource eine JPA-Entität ist, in welcher Tabelle sie persistiert wird sowie welche Spalten diese Tabelle besitzen soll.

²³<https://spring.io/projects/spring-data-jpa#overview>

²⁴<https://hibernate.org/>

Die übrigen Annotations stammen aus der Bibliothek Starter Validation²⁵ von Spring Boot, welche Hibernate Validator²⁶, eine Implementation der Jakarta Bean Validation API²⁷, zu dem Projekt hinzufügt. Durch die Annotations wie zum Beispiel @Pattern werden die Eigenschaften eines Buches beim Aufruf eines Endpunktes automatisch validiert. Dies wird im Abschnitt zur Validierung genauer erläutert.

Abgesehen von den Ressourcen werden auch DTOs eingesetzt, insbesondere um die Daten in einer Anfrage an einen Endpunkt oder einer Antwort von einem Endpunkt zu spezifizieren. Diese wurden als Records definiert, da ihre Eigenschaften unveränderbar sind und lediglich ihr Typ, Name und gegebenenfalls Bedingungen über Annotations festgelegt werden sollen. In Java verhalten sich Records sehr ähnlich wie in C#, weshalb nähere Informationen zu diesen in Kapitel 4.1.2 nachgelesen werden können.

Abruf eines Objekts

Nun soll der erste Endpunkt für die Ressource Book erstellt werden. Hierfür stellt Spring Data JPA verschiedene, als Repositories bezeichnete Interfaces bereit, die vordefinierte Methoden für die wichtigsten Operationen besitzen. Eines dieser Interfaces ist JpaRepository<T, ID>. Es erweitert das ListCrudRepository<T, ID>, das wie der Name vermuten lässt, Methoden für das Erstellen, Lesen, Aktualisieren und Löschen von Ressourcen anbietet, sowie das ListPagingAndSortingRepository<T, ID>, das zusätzlich Pagination und Sortierung ermöglicht. Die beiden Generics stehen für den Entitätstyp sowie den Typ der Id der Entität. Zudem ist das JpaRepository<T, ID> insbesondere für die Nutzung mit einer relationalen Datenbank wie in dieser Arbeit MariaDB geeignet [13, 30]. Aus diesen Gründen wird für den Webservice ein Interface BooksRepository geschrieben, das die Interfaces JpaRepository sowie JpaSpecificationExecutor<T> erweitert. Auf Letzteres wird später eingegangen.

```

1  @Repository
2  public interface BooksRepository extends JpaRepository<Book, Integer>,
   JpaSpecificationExecutor<Book> {}
```

Listing 23: Java-Code des BooksRepositorys

²⁵<https://central.sonatype.com/artifact/org.springframework.boot/spring-boot-starter-validation/3.0.3>

²⁶<https://hibernate.org/validator/>

²⁷<https://beanvalidation.org/>

Das Repository wird schließlich von einem sogenannten Controller, der im zu entwickelnden RESTful Webservice BooksController heißt, genutzt, welcher für das Bearbeiten von HTTP-Anfragen zuständig ist. Dieser ist im unten stehenden Listing zu sehen. Hierbei ist nur die Methode abgebildet, welche tatsächlich für die aktuell beschriebene Aufgabe, das Abrufen eines Buches mithilfe seiner Id, benötigt wird.

```

1  @RestController
2  @RequestMapping( "/v1/books" )
3  @Validated
4  public class BooksController {
5      private final BooksRepository booksRepository;
6
7      @Autowired
8      public BooksController(BooksRepository booksRepository) {
9          this.booksRepository=booksRepository;
10     }
11
12     @GetMapping( "{id}" )
13     public ResponseEntity<Book> getBook(@PathVariable int id) throws
14         NotFoundException {
15         Book book=booksRepository.findById(id).orElseThrow(() -> new
16             NotFoundException("Book not found with id " + id));
17         return ResponseEntity.ok().body(book);
18     }
19     // weitere Methoden
20 }
```

Listing 24: Java-Code des BooksController mit der Methode getBook

Wie das Listing zeigt, wird für das Abrufen des Buches die vorgefertigte Methode `findById` des Repositories genutzt. Wurde ein Buch gefunden, wird es gewrapped in der generischen Klasse `ResponseEntity<T>` zurückgegeben. Diese repräsentiert eine HTTP-Antwort, deren Statuscode, Headers und Body über Methoden mithilfe des Builder-Patterns gesetzt werden können. Was im Fehlerfall passiert, wird an späterer Stelle erläutert.

Es ist anzumerken, dass die Annotations in den beiden Listings nicht nur als Information, sondern teilweise, beispielsweise im Falle von `@Autowired`, auch zur Dependency Injection

dienen. Die Annotation `@GetMapping` über der Methode in Verbindung mit `@RequestMapping` über der Klassendefinition des BooksControllers stellt zudem sicher, dass nur Anfragen, die das HTTP-Verb GET und den Pfad „v1/books/id“ verwenden, zu einer Ausführung der Methode `getBook` führen. Entsprechend gibt es auch Annotations für die anderen üblichen HTTP-Verben. Über `@PathVariable` wird darüber hinaus ein Methodenparameter, in diesem Fall `id`, an eine Variable aus der URL gebunden. Entsprechend gibt es auch `@RequestParam` für Query-Parameter und `@RequestBody` für den Body der Anfrage.

Abruf aller Objekte mit Pagination, Sortierung und Filterung

Das Abrufen aller Bücher wird grundsätzlich mit der Methode `findAll` des JpaRepositories implementiert. Dabei ist jedoch zu beachten, dass bei diesem Endpunkt zum einen auch Pagination und eine aufsteigende Sortierung nach der Buch-Id umgesetzt und zum anderen optional nach einem Suchbegriff gefiltert werden können soll. Was Ersteres betrifft, so hat das JpaRepository durch Erweiterung des Interfaces `ListPagingAndSortingRepository<T, ID>` eine Überladung der Methode `findAll`, der ein Objekt der Klasse `PageRequest` übergeben werden kann. Dieses ermöglicht die Angabe einer Seite, der Anzahl der Einträge pro Seite und einer Sortierung in Form eines Objektes der Klasse `Sort`. Soll hingegen ein Suchbegriff verwendet werden, so wird eine weitere Überladung der Methode `findAll` benötigt. Diese wird durch das Interface `JpaSpecificationExecutor<T>`, das ebenso von dem BooksRepository erweitert wird, bereitgestellt und empfängt neben dem `PageRequest` ein Objekt einer Klasse, welche das Interface `Specification<T>` implementiert. Hierfür wurde die Klasse `SearchBooksSpecification` geschrieben. Sie überschreibt die abstrakte Methode `toPredicate` des Interfaces, die im Wesentlichen eine WHERE-Klausel für die SQL-Abfrage erstellt, welche in `findAll` ausgeführt wird. Die Überladungen der Methode `findAll` liefern letztendlich eine `Page`, über die auf die maximal abrufbare Seite sowie die Bücherliste zugegriffen werden kann. Beides wird in Form eines `BooksResponse`-Objekts, das in eine `ResponseEntity` gewrapped ist, von der Controller-Methode `getAllBooks` zurückgegeben.

```

1  @GetMapping( "")
2  public ResponseEntity<BooksResponse> getAllBooks(@RequestParam @Min(value=1,
3      message="page must be at least 1.") int page, @RequestParam @Min(value=1,
4      message="per_page must be at least 1.") @Max(value=50, message="per_page must
5      be less than or equal to 50") int perPage)
6  {
7      return new ResponseEntity<BooksResponse>(booksService.getAllBooks(page, perPage),
8          HttpStatus.OK);
9  }

```

```

1    be at most 50.") int per_page, @RequestParam(required=false) @Size(min=1,
2    max=30, message="query must have between {min} and {max} characters.") String
3    query) {
4
5    Page<Book> books=null;
6
7    if(query!=null) {
8
9      Specification<Book> specification=new SearchBooksSpecification(new String[]{"
10      "isbn", "title", "author"}, query);
11
12      books=booksRepository.findAll(specification, PageRequest.of(page-1, per_page,
13          Sort.by(new Sort.Order(Sort.Direction.ASC, "id"))));
14
15    } else {
16
17      books=booksRepository.findAll(PageRequest.of(page-1, per_page, Sort.by(new
18          Sort.Order(Sort.Direction.ASC, "id"))));
19
20    }
21
22
23    return ResponseEntity.ok().body(new BooksResponse(books.getTotalPages(), books.
24        getContent()));
25  }

```

Listing 25: Java-Code der Controller-Methode getAllBooks

Erstellung eines Objekts

Um ein Buch zu erstellen, stellt das JpaRepository die Methode saveAndFlush bereit, der ein neues Book-Objekt übergeben wird. Sie liefert das erzeugte Buch zurück, das die automatisch generierte Id enthält. Dadurch kann die Id von der createBook Methode des Controllers gewrappt in eine ResponseEntity zurückgegeben werden.

```

1  @PostMapping("")
2  public ResponseEntity<IdResponse> createBook(@RequestBody @Valid CreateBook
3      createBook) {
4
5      Book newBook=booksRepository.saveAndFlush(new Book(createBook.isbn(),
6          createBook.title(), createBook.author()));
7
8      return ResponseEntity.status(HttpStatus.CREATED).body(new IdResponse(newBook.
9          getId()));
10  }

```

Listing 26: Java-Code der Controller-Methode createBook

Änderung einzelner Attribute eines Objekts

Für das Aktualisieren eines Buches gibt es keine direkte Methode in den beiden Repository Interfaces, welche BooksRepository erweitert. Stattdessen muss das Buch mit findById abgerufen, die gewünschten Eigenschaften modifiziert und anschließend mit saveAndFlushpersistiert werden.

```

1 @PatchMapping( "/{id}" )
2 public ResponseEntity<?> updateBook(@PathVariable int id, @RequestBody @Valid
3                                     UpdateBook createBook) throws NotFoundException {
4     Book book=booksRepository.findById(id).orElseThrow(() -> new NotFoundException(
5         "Book not found with id " + id));
6     if(createBook.isbn() != null) {
7         book.setIsbn(createBook.isbn());
8     }
9     if(createBook.title() != null) {
10        book.setTitle(createBook.title());
11    }
12    if(createBook.author() != null) {
13        book.setAuthor(createBook.author());
14    }
15    booksRepository.saveAndFlush(book);
16    return ResponseEntity.noContent().build();
17 }
```

Listing 27: Java-Code der Controller-Methode updateBook

Löschung eines Objekts

Der letzte Endpunkt soll das Löschen eines Buches ermöglichen. Dazu wird mit findById geprüft, ob das Buch mit der gewünschten Id vorhanden ist. Trifft dies zu, so kann es mit der Methode deleteById des Interfaces JpaRepository entfernt werden.

```

1 @DeleteMapping( "/{id}" )
2 public ResponseEntity<?> deleteBook(@PathVariable int id) throws
3                                     NotFoundException {
4     booksRepository.findById(id).orElseThrow(() -> new NotFoundException("Book not
5         found with id " + id));
```

```

4   booksRepository.deleteById(id);
5   return ResponseEntity.noContent().build();
6 }
```

Listing 28: Java-Code der Controller-Methode deleteBook

Validierung von erhaltenen Daten

Weiterhin soll darauf eingegangen werden, auf welche Weise die Attribute der Ressourcen und DTOs sowie auch Query-Parameter und Request-Bodies validiert werden.

Bezüglich der ersten drei wurde unter anderem in Listing 22 und Listing 24 gezeigt, dass durch die Bibliothek Starter Validation von Spring Boot zahlreiche Annotations von Hibernate Validator zur Verfügung stehen, mit denen für die Eigenschaften einer Klasse oder eines Records oder für einen Query-Parameter Bedingungen definiert werden können. So kann z. B. für die ISBN mithilfe von @Pattern eine Regular Expression festgelegt werden oder für den Titel über @Size eine minimale und maximale Textlänge. Für jede dieser Annotations wird neben speziellen Werten, wie „min“ bei @Size, auch eine „message“ definiert, die für die Fehlerbehandlung im nächsten Abschnitt bedeutend ist. Zudem muss darauf hingewiesen werden, dass der verwendete BooksController die Annotation @Validated vor seiner Klassendefinition besitzen muss, damit Spring die Methodenparameter validiert.

Was hingegen komplexe Typen wie die Request-Bodies betrifft, so kann gesagt werden, dass dem Framework durch die Angabe von @Valid vor dem Übergabeparameter mitgeteilt wird, dass das Argument durch Hibernate Validator geprüft werden soll, bevor der Inhalt der Methode ausgeführt wird (siehe zum Beispiel Listing 26).

Insgesamt bringt Spring Boot folglich alles Notwendige für die Validierung mit, wodurch diese keine große Herausforderung mehr darstellt.

Behandlung von Fehlern

Zur Fehlerbehandlung wurde eine Klasse ErrorHandler geschrieben. Ihre Annotation @ControllerAdvice macht sie zu einer einzigen globalen Fehlerbehandlungs-Komponente, deren Methoden allen Controllern des Webservices zur Verfügung stehen. Jede der Methoden erhält die Annotation @ExceptionHandler, bei der in Klammern angegeben wird, auf welche Art von Exception sie reagiert. Neben automatisch geworfenen Exceptions wie Cons-

straintViolationException bei der Validierung primitiver Typen werden auch eine eigens geschriebene NotFoundException, die von Exception erbt, sowie eine allgemeine Exception unbekannten Typs durch eine solche Methode behandelt. Alle Handler-Methoden geben eine ResponseEntity mit dem passenden Statuscode sowie einem Objekt des Typs ErrorResponse oder ErrorsResponse im Body zurück. Die jeweilige „message“ beziehungsweise die „messages“ wurden wie in dem vorherigen Abschnitt beschrieben, teilweise angepasst, um für den Aufrufenden verständlich zu sein.

```

1  @ControllerAdvice
2  public class ErrorHandler {
3      @ExceptionHandler(NotFoundException.class)
4      public ResponseEntity<?> notFoundHandler(NotFoundException exception) {
5          return new ResponseEntity<>(new ErrorResponse(exception.getMessage()) ,
6              HttpStatus.NOT_FOUND);
7      }
8
9      @ExceptionHandler({ ConstraintViolationException.class })
10     public ResponseEntity<?> badRequestHandler(ConstraintViolationException
11         exception) {
12         List<String> errorMessages=new ArrayList<String>();
13         for(ConstraintViolation<?> constraintViolation: exception.
14             getConstraintViolations()) {
15             errorMessages.add(constraintViolation.getMessage());
16         }
17         return new ResponseEntity<>(new ErrorsResponse(errorMessages) , HttpStatus.
18             BAD_REQUEST);
19     }
20     // Weitere Handler-Methoden
21 }
```

Listing 29: Java-Code des ErrorHandlers

Konfiguration des Servers

Was die Konfiguration betrifft, so wird diese zu großen Teilen bereits von dem Tool Spring Initializr übernommen. Hierbei kann entschieden werden, welche Java und Spring Boot Versionen, welches Build-Werkzeug sowie welche Spring Module verwendet werden sollen.

Ebenso werden Metadaten wie zum Beispiel der Anwendungsname angegeben. All diese Einstellungen sind im Nachhinein selbstverständlich noch änderbar. Für den Webservice in dieser Arbeit wurde Gradle²⁸ als Build-Tool ausgesucht. In diesem Fall wird automatisch eine Datei build.gradle angelegt, welche insbesondere die Abhängigkeiten der Applikation inklusive ihrer Versionen beinhaltet.

```

1 plugins {
2     id 'java'
3     id 'org.springframework.boot' version '3.0.2'
4     id 'io.spring.dependency-management' version '1.1.0'
5 }
6
7 group = 'de'
8 version = '0.0.1-SNAPSHOT'
9 sourceCompatibility = '19'
10
11 repositories {
12     mavenCentral()
13 }
14
15 dependencies {
16     implementation 'org.springframework.boot:spring-boot-starter-web'
17     implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
18     implementation 'org.springframework.boot:spring-boot-starter-validation'
19     runtimeOnly 'org.mariadb.jdbc:mariadb-java-client'
20 }
```

Listing 30: Groovy-Code der Datei build.gradle

Der Projektname hingegen ist in der ebenfalls erzeugten Datei settings.gradle zu finden.

```

1 rootProject.name = 'libraryapi'
```

Listing 31: Groovy-Code der Datei settings.gradle

²⁸<https://gradle.org/>

Darüber hinaus gibt es noch eine Konfigurationsdatei für Konstanten, die per Konvention application.properties heißt. Sie beinhaltet Schlüssel-Wert-Paare, z. B. die Konfigurationsdaten einer Datenbank, welche automatisch von Spring Boot erkannt und verwendet werden.

```

1 ## MariaDB
2 spring.datasource.url=jdbc:mariadb://localhost:28374/libraryapi
3 spring.datasource.username=libraryapi
4 spring.datasource.password=1234
5 spring.datasource.driver-class-name=org.mariadb.jdbc.Driver
6 spring.datasource.tomcat.max-active=150
7
8 ## Hibernate
9 spring.jpa.generate-ddl=true
10 spring.jpa.hibernate.ddl-auto=update

```

Listing 32: Datei application.properties des Spring Boot-Projekts

Zuletzt wird der Einstiegspunkt der Applikation betrachtet, die Klasse LibraryApiApplication. Auch sie wird bei der Anwendung des Spring Initializrs automatisch angelegt und beinhaltet eine Main-Methode, in der die Methode SpringApplication.run() von Spring Boot aufgerufen wird. Die Klasse wird zudem mit der Annotation @SpringBootApplication versehen, welche unter anderem für die automatische Konfiguration, z. B. das Erkennen einer mit @RestController markierten Komponente, durch Spring Boot sorgt.

```

1 @SpringBootApplication
2 public class LibraryApiApplication {
3     public static void main(String[] args) {
4         SpringApplication.run(LibraryApiApplication.class, args);
5     }
6 }

```

Listing 33: Java-Code der Datei LibraryApiApplication.java

4.2.3 Bewertung

Isabell Waas

Hiermit wurde der RESTful Webservice mit Java und Spring Boot vollständig entwickelt und kann bewertet werden.

Dokumentation und Wartung

Um sich mit Spring auseinanderzusetzen, wird als Erstes dessen Webseite²⁹ besucht. Ein Blick auf die Navigationsleiste zeigt, dass diese verschiedenste Wege anbietet, damit sich möglichst jeder Entwickler auf die von ihm präferierte Art und Weise über Spring informieren kann. Zu den wichtigsten Menüpunkten gehört „Learn“. Dort ist insbesondere ein Quickstart Guide zu finden sowie Guides und Tutorials zu Themen wie „Building a RESTful Web Service“ oder „Accessing Data with JPA“, deren Dauer von 15 Minuten bis hin zu drei Stunden variieren. So kann der Anwender individuell entscheiden, in welcher Geschwindigkeit und welchem Umfang er ein Thema erlernen möchte. Unter „Academy“ gibt es dagegen eine Webseite mit einigen Videokursen, was gut für Personen ist, die Videos gegenüber Text bevorzugen. Abgesehen davon ist der Menüpunkt „Projects“ hervorzuheben, dessen Übersichtsseite die Abbildung unten zeigt.

The screenshot shows the official Spring website's "Projects" section. At the top, there is a navigation bar with links for "Why Spring", "Learn", "Projects", "Academy", "Support", and "Community". Below the navigation, the title "Projects" is displayed in large, bold, black font. A subtext explains that the site provides projects for various needs, from configuration to security, web apps to big data. The main content area is divided into four cards, each representing a Spring module:

- Spring Boot**: Takes an opinionated view of building Spring applications and gets you up and running as quickly as possible.
- Spring Framework**: Provides core support for dependency injection, transaction management, web apps, data access, messaging, and more.
- Spring Data**: Provides a consistent approach to data access – relational, non-relational, map-reduce, and beyond.
- Spring Cloud**: Provides a set of tools for common patterns in distributed systems. Useful for building and deploying microservices.

Abbildung 27: Übersichtsseite der Module von Spring [11]

²⁹<https://spring.io/>

Die Seite stellt sämtliche Module von Spring übersichtlich dar und beschreibt diese in wenigen Worten. Klickt der Nutzer auf ein Modul, erhält er auf einer verlinkten Seite noch mehr Informationen zu diesem. Ebenso sind dort für jedes Modul eine eigene Reference Documentation und eine Api-Dokumentation zu finden. Erstere erläutert wichtige Themen und Funktionen des jeweiligen Moduls hauptsächlich in Textform unter Verwendung vieler Codebeispiele. Diese sind in der Regel sowohl in Java als auch in Kotlin abgebildet, da Spring mit beiden Sprachen genutzt werden kann. Das Bildschirmfoto unten zeigt die Reference Documentation von Spring Boot.

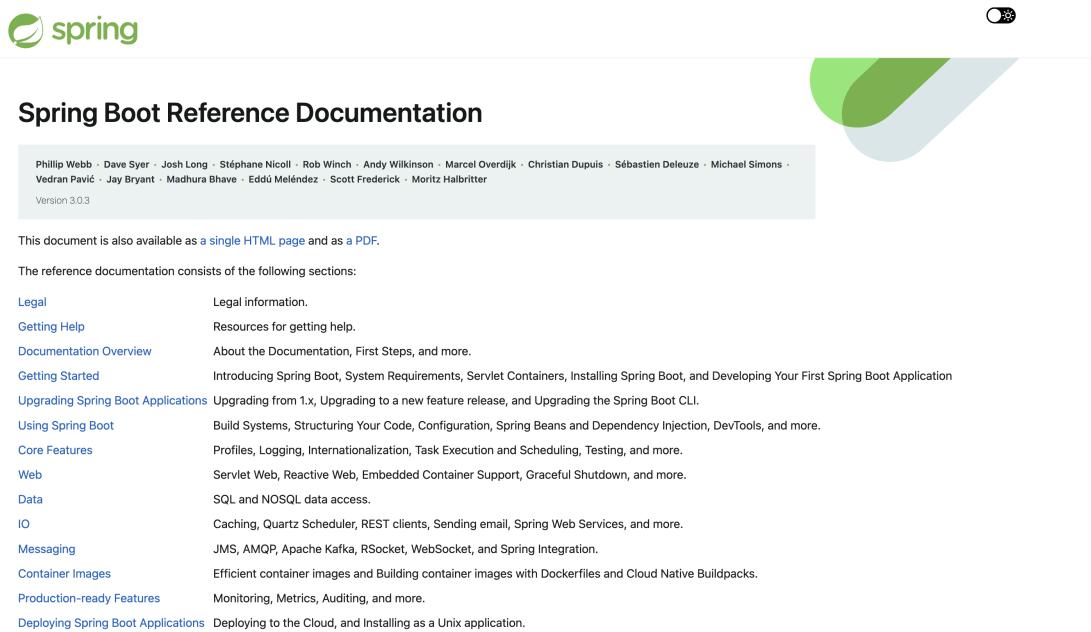
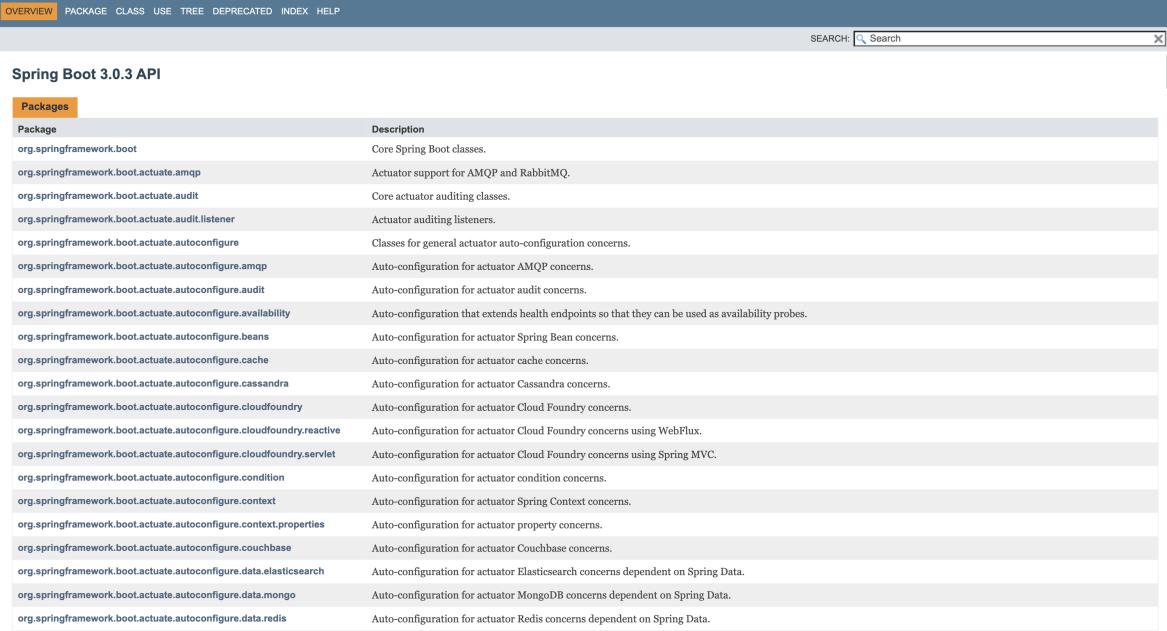


Abbildung 28: Reference Documentation von Spring Boot [29]

Bei der Api-Dokumentation handelt es sich dagegen eher um eine Bibliothek, die insbesondere alle Klassen und Methoden im Detail beleuchtet. Wer die Dokumentation von Java kennt, wird sich in den Api-Dokumentationen der Spring-Module sofort zurechtfinden, da diese identisch aussehen und aufgebaut sind. Im Folgenden ist die Api-Dokumentation von Spring Boot 3.0.3 abgebildet.



The screenshot shows the Spring Boot 3.0.3 API documentation interface. At the top, there's a navigation bar with links for OVERVIEW, PACKAGE, CLASS, USE, TREE, DEPRECATED, INDEX, and HELP. A search bar is located at the top right. Below the navigation, the title "Spring Boot 3.0.3 API" is displayed. Underneath the title, there's a heading "Packages". A table follows, listing various package names under the "Package" column and their descriptions under the "Description" column.

Package	Description
org.springframework.boot	Core Spring Boot classes.
org.springframework.boot.actuate.amqp	Actuator support for AMQP and RabbitMQ.
org.springframework.boot.actuate.audit	Core actuator auditing classes.
org.springframework.boot.actuate.audit.listener	Actuator auditing listeners.
org.springframework.boot.actuate.autoconfigure	Classes for general actuator auto-configuration concerns.
org.springframework.boot.actuate.autoconfigure.amqp	Auto-configuration for actuator AMQP concerns.
org.springframework.boot.actuate.autoconfigure.audit	Auto-configuration for actuator audit concerns.
org.springframework.boot.actuate.autoconfigure.availability	Auto-configuration that extends health endpoints so that they can be used as availability probes.
org.springframework.boot.actuate.autoconfigure.beans	Auto-configuration for actuator Spring Bean concerns.
org.springframework.boot.actuate.autoconfigure.cache	Auto-configuration for actuator cache concerns.
org.springframework.boot.actuate.autoconfigure.cassandra	Auto-configuration for actuator Cassandra concerns.
org.springframework.boot.actuate.autoconfigure.cloudfoundry	Auto-configuration for actuator Cloud Foundry concerns.
org.springframework.boot.actuate.autoconfigure.cloudfoundry.reactive	Auto-configuration for actuator Cloud Foundry concerns using WebFlux.
org.springframework.boot.actuate.autoconfigure.cloudfoundry.servlet	Auto-configuration for actuator Cloud Foundry concerns using Spring MVC.
org.springframework.boot.actuate.autoconfigure.condition	Auto-configuration for actuator condition concerns.
org.springframework.boot.actuate.autoconfigure.context	Auto-configuration for actuator Spring Context concerns.
org.springframework.boot.actuate.autoconfigure.context.properties	Auto-configuration for actuator property concerns.
org.springframework.boot.actuate.autoconfigure.couchbase	Auto-configuration for actuator Couchbase concerns.
org.springframework.boot.actuate.autoconfigure.data.elasticsearch	Auto-configuration for actuator Elasticsearch concerns dependent on Spring Data.
org.springframework.boot.actuate.autoconfigure.data.mongo	Auto-configuration for actuator MongoDB concerns dependent on Spring Data.
org.springframework.boot.actuate.autoconfigure.data.redis	Auto-configuration for actuator Redis concerns dependent on Spring Data.

Abbildung 29: Api-Dokumentation von Spring Boot 3.0.3 [12]

Insgesamt kann festgestellt werden, dass die Webseite von Spring zahlreiche Wege sowohl für Anfänger als auch für Fortgeschrittene eröffnet, um das Framework näher zu bringen. Dennoch muss gesagt werden, dass gerade Springs Modularität, welche zu seinen größten Stärken zählt, den Einstieg in das Framework erschwert. Besonders bei der Entwicklung mit mehreren Modulen kommt es immer wieder vor, dass sich Informationen zu ähnlichen Themen in verschiedenen Dokumentationen und Guides mühevoll zusammengesucht werden müssen. Dies verlangsamt die Programmierung und veranlasst den Nutzer dazu, trotz der umfassenden Ressourcen auf der Webseite von Spring immer wieder auf einfachere, auf die aktuell zu lösende Aufgabe abgestimmte externe Quellen zurückzugreifen. Angesichts der Tatsache, dass Spring ein komplexes und über die Zeit stark gewachsenes Framework ist, sind die Webseite und verlinkten Dokumentationen jedoch auf keinen Fall schlecht.

Zuletzt soll noch auf die Wartung von Spring Boot eingegangen werden. Für jedes Modul, so also auch für Spring Boot, ist auf der Webseite von Spring pro Release dargestellt, in welchem Zeitraum er gewartet wird. Die Grafik zeigt den Supportzeitraum für die Versionen von Spring Boot, wobei kleinere Updates wie zum Beispiel von 3.0.2 auf 3.0.3 nicht mit aufgeführt sind. Ebenso ist zu erkennen, dass Spring Boot etwa ein- bis zweimal im Jahr eine neue Version erhält und damit stets aktuell ist. Dabei erhält jeder Release etwa ein Jahr

Support, welcher Sicherheitsupdates und Bugfixes beinhaltet, sowie etwa zwei Jahre und ein paar Monate kommerziellen Support. Letzterer stammt im Gegensatz zu dem einfachen Support von Spring Experten und nicht von der Community und ist wesentlich länger verfügbar [11]. Insgesamt kann die Wartung von Spring Boot damit als sehr gut bewertet werden.

The screenshot shows the official Spring Boot website at spring.io/projects/spring-boot. The top navigation bar includes links for Why Spring, Learn, Projects, Academy, Support, and Community. The main content area is titled "Spring Boot 3.0.3". Below the title, there are tabs for OVERVIEW, LEARN, SUPPORT (which is selected), and SAMPLES. The SUPPORT tab displays a table of releases:

Branch	Initial Release	End of Support	End Commercial Support *
3.1.x	2023-05-18	2024-05-18	2025-08-18
3.0.x	2022-11-24	2023-11-24	2025-02-24
2.7.x	2022-05-19	2023-11-18	2025-02-18
2.6.x	2021-11-17	2022-11-24	2024-02-24
2.5.x	2021-05-20	2022-05-19	2023-08-24
2.4.x	2020-11-12	2021-11-18	2023-02-23
2.3.x	2020-05-15	2021-05-20	2022-08-20
2.2.x	2019-10-16	2020-10-16	2022-01-16
2.1.x	2018-10-30	2019-10-30	2021-01-30
2.0.x	2018-03-01	2019-03-01	2020-06-01
1.5.x	2017-01-30	2019-08-06	2020-11-06

Abbildung 30: Releases von Spring Boot [11]

Lines of Code

Zum Ermitteln der Lines of Code mit dem Tool `cloc` wurden nur Java-Dateien berücksichtigt. Nur diese beinhalten tatsächlichen Code für den Webservice und dienen nicht anderen Zwecken, wie z. B. der Konfiguration des Servers. Die Werte, welche die Messung mit `cloc` ergab, sind in der folgenden Tabelle visualisiert.

Language	files	blank	comment	code
Java	13	48	0	237

Tabelle 4: Ausgabe des Programms `cloc` bei der Implementierung mit Java und Spring Boot

Notwendigkeit zusätzlicher Bibliotheken

In dem Webservice wird Spring, insbesondere seine Module Spring Boot und Spring Data JPA verwendet. Letzteres wird durch die Bibliothek Starter Data JPA³⁰ von Spring Boot eingebunden und nutzt das Open-Source ORM-Tool Hibernate, welches eine Implementierung von JPA ist. Hibernate ist grundsätzlich unabhängig von Spring, wird jedoch standardmäßig von Spring Data JPA eingesetzt. Spring Data JPA bietet außerdem durch seine Repositories für die gängigsten Datenbankoperationen bereits vorgefertigte Methoden an, sodass SQL-Anweisungen kaum mehr selbst geschrieben werden müssen und die Verwendung einer Datenbank mit Spring insgesamt mit wenig Code möglich ist.

Darüber hinaus wird die Bibliothek Starter Validation von Spring Boot genutzt. Diese fügt Hibernate Validator, eine Implementierung der Java Bean Validation API zu dem Projekt hinzu, über die auch die Validierung verschiedenster Daten sehr einfach gestaltet wird. Wie auch das ORM ist der Validator von Hibernate nicht auf die Verwendung mit Spring beschränkt und kann auch in Zusammenhang mit anderen Frameworks eingesetzt werden.

Abgesehen davon wird noch die Bibliothek Starter Web³¹ von Spring Boot in den Abhängigkeiten in der Datei build.gradle (siehe Listing 30) aufgeführt. Diese ermöglicht die Entwicklung von Webapplikationen und nutzt unter anderem Apache Tomcat³² als Webserver beziehungsweise -container, der auch für Spring-unabhängige Zwecke einsetzbar ist.

Zuletzt wird der Mariadb Connector/J³³ als JDBC Treiber für die Datenbank eingesetzt. Dieser dient allgemein dazu, in Java Programmen mit MariaDB oder MySQL Datenbanken zu kommunizieren und ist nicht an Spring gebunden.

³⁰<https://central.sonatype.com/artifact/org.springframework.boot/spring-boot-starter-data-jpa/3.0.3>

³¹<https://central.sonatype.com/artifact/org.springframework.boot/spring-boot-starter-web/3.0.3>

³²<https://tomcat.apache.org/>

³³<https://central.sonatype.com/artifact/org.mariadb.jdbc/mariadb-java-client/3.1.2/overview>

Wichtige Architekturprinzipien und besondere Sprachfunktionen

Bei der Bewertung des Frameworks Spring Boot sollen auch die Architektur und besondere Sprachfeatures betrachtet werden. Dazu wird in dem Schaubild unten der Projektaufbau veranschaulicht, wobei nur Ordner mit Java-Dateien sowie die wichtigsten Dateien abgebildet werden.

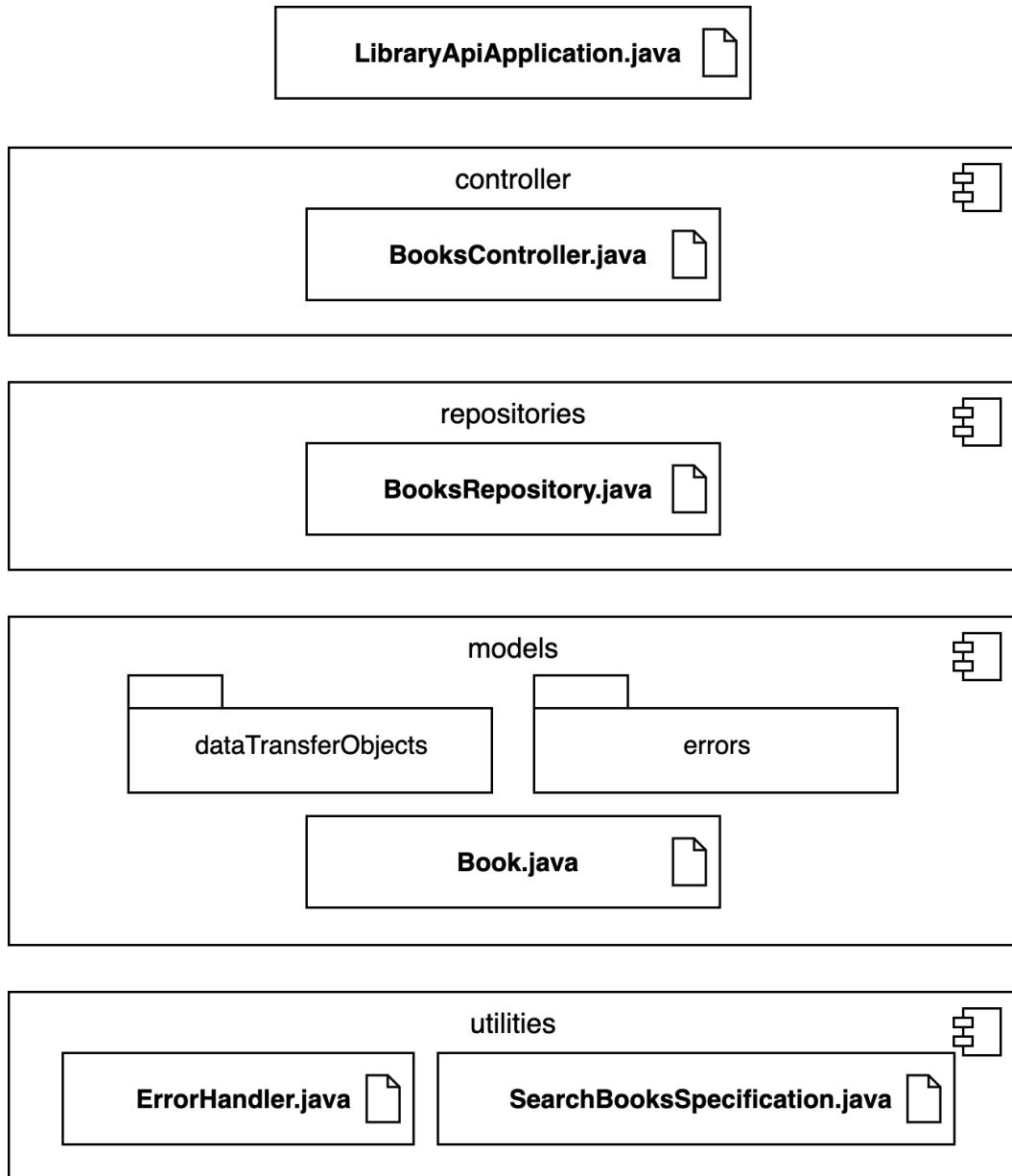


Abbildung 31: Architektur des Webservices mit Spring Boot

Ein besonderes Augenmerk soll auf die Aufteilung der Funktionalität in gekapselte Einheiten und die Einhaltung des Single-Responsibility-Prinzips durch diese gelegt werden. Spring Boot ermöglicht beides sehr gut. Im Folgenden sollen die Ordner und Dateien in Abbildung 31 erläutert werden, wobei zusammengehörige gemeinsam beschrieben werden.

Bei der Datei LibraryApiApplication.java, dem Startpunkt des Webservices mit der Main-Methode handelt es sich um die eigentliche Spring Boot Anwendung, da diese über eine Annotation als solche gekennzeichnet wird. Zu erwähnen ist, dass Annotations ein Sprachfeature von Java sind und Metadaten darstellen. Auch in den anderen Einheiten des Projektes kommen zahlreiche Annotations zum Einsatz, was zeigt, dass Spring Boot stark auf diese setzt. Viele von ihnen stellen Informationen bereit, welche für eines der Hauptfeatures von Spring Boot, die automatische Konfiguration durch Dependency Injection, notwendig sind. Diese reduziert den Entwicklungsaufwand und die Fehleranfälligkeit, da Komponenten nicht manuell initialisiert und an sie benötigende Klassen übergeben werden müssen. Statt dessen erkennt Spring Boot ihre Funktion mithilfe einer Annotation und kümmert sich um alles Weitere.

Die Klasse Controller wird mit einer Annotation @RestController versehen, was bereits auf ihre Aufgabe hindeutet. Denn in ihr werden die Endpunkte des RESTful Webservices in Form von Methoden implementiert. Über Annotations wie @RequestMapping auf Klasse- und @GetMapping auf Methodenebene werden die Pfade und HTTP-Verben der Endpunkte festgelegt. Somit kann gesagt werden, dass jeder Controller einen Hauptpfad, z. B. „/v1/books“, und jede seiner Methoden einen Unterpfad wie „/{id}“ sowie eine HTTP-Methode repräsentiert. Spring Boot erlaubt es, jederzeit weitere Controller und Endpunkte zu ergänzen, wobei, wenn diese die genannten Annotations besitzen, durch Dependency Injection im Startpunkt der Anwendung nichts verändert werden muss. Es kann damit gesagt werden, dass Routergruppen durch Controllerklassen gekapselt sind, wobei Spring Boot diese Kapselung durch die Klassen-Annotations fördert.

Datenbankabfragen müssen dank Spring Boot kaum mehr geschrieben werden, denn dieses stellt verschiedene Repository-Interfaces bereit, welche für die häufigsten Operationen vorgefertigte Methoden wie findByID besitzen. Folglich muss lediglich für jede zu verwaltende Model-Klasse ein Interface geschrieben werden, das mit @Repository markiert ist und

eines oder mehrere der vorgefertigten Repository-Interfaces erweitert. Wie viele solcher Interfaces zum Einsatz kommen, steht dem Entwickler frei. Für Anfragen, die eine komplexe WHERE-Bedingung erfordern, steht zudem das Interface Specification<T> mit seiner Methode toPredicate zur Verfügung, weshalb auch hierfür kein SQL geschrieben werden muss.

Die meisten Model-Klassen werden als Records erstellt. Das Java-Feature dient zur Definition von unveränderbaren Klassen, bei denen notwendiger Boilerplate-Code automatisch erzeugt wird, was viel Schreibarbeit spart. Die in der Datenbank verwalteten Models müssen allerdings als normale Klassen erstellt werden, da ihre Attribute veränderbar sein müssen. Ebenso erhalten sie die Annotation @Entity. Auch Fehlerklassen sind keine Records, da sie von der Klasse Exception erben müssen.

Die Fehlerbehandlung findet an einem zentralen Ort, dem ErrorHandler, statt. Er beinhaltet Methoden, die per Annotation auf verschiedene Fehlerarten reagieren. Dabei kann er beim Hinzufügen weiterer Fehlerarten problemlos um weitere Methoden ergänzt werden.

Insgesamt nimmt Spring Boot dem Entwickler durch Interfaces wie die Repositories sowie zahlreiche Annotations, die Dependency Injection, aber auch Validierung verschiedenster Datentypen sehr einfach gestalten, viel Arbeit ab. Durch die gute Kapselung von Verantwortlichkeiten sind Erweiterungen ohne weitreichende Änderungen am bestehenden Code möglich. Was die Sprachfeatures betrifft, so unterstützt Java die Entwicklung durch die erwähnten Annotations, aber auch durch Records. Leider können diese jedoch nicht überall eingesetzt werden, weshalb noch immer teilweise viel Boilerplate-Code notwendig ist.

Performance

Auch bei dem Webservice mit Java und Spring Boot soll nun die Performance mit dem Tool Artillery und dem selbst geschriebenen Testskript gemessen werden. In dem anschließend generierten Report wurden wieder die beiden Diagramme auf der folgenden Seite genauer betrachtet.

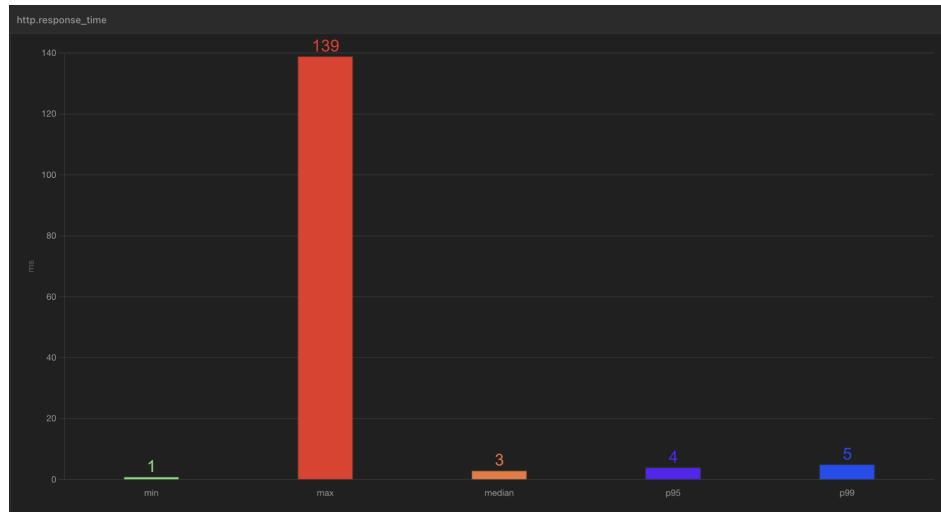


Abbildung 32: Ausgewählte Antwortzeiten für Java mit Spring Boot (Balkenwerte manuell hinzugefügt)

In der oben abgebildeten Grafik werden ausgewählte Antwortzeiten des Webservices dargestellt. Unter diesen befinden sich die insgesamt minimale und maximale Antwortdauer sowie die Maximaldauern für 50%, 95% und 99% der Anfragen. Das nun folgende Bildschirmfoto aus dem Report zeigt hingegen die Visualisierung ausgewählter Sessionlängen, wobei wieder dieselben Säulenarten wie zuvor eingesetzt werden.

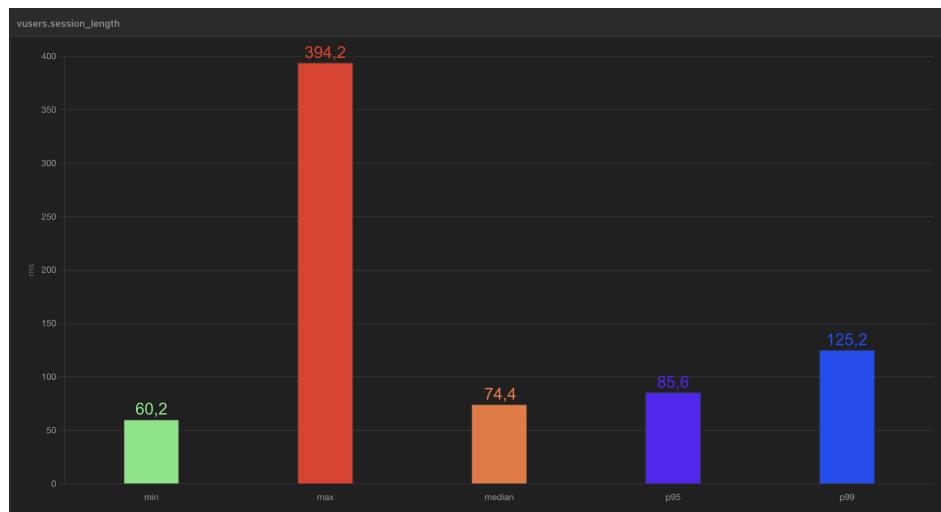


Abbildung 33: Ausgewählte Sessionlängen für Java mit Spring Boot (Balkenwerte manuell hinzugefügt)

4.3 TypeScript mit Express.js

4.3.1 Einführung in die Sprache und das Framework

Isabell Waas



Abbildung 34: TypeScript Logo [38]



Abbildung 35: Express.js Logo [23]

Die 2012 von Microsoft entwickelte Skriptsprache TypeScript³⁴ ist eine der beiden jüngsten Sprachen, welche im Rahmen dieser Arbeit verwendet wird. Bei ihr handelt es sich um eine Obermenge der wesentlich älteren und insbesondere aus dem Web nicht mehr wegzudenkenden Sprache JavaScript. Wie ihr Name schon vermuten lässt, ist der Hauptunterschied von TypeScript zu JavaScript ihre Typisierung. TypeScript ermöglicht es, den Programmcode sowohl statisch als auch dynamisch zu typisieren und so klareren und stabileren Code zu erzeugen, da viele Fehler schon vor der Laufzeit erkannt werden. Zudem wird TypeScript Code transpiliert, was bedeutet, er wird in eine Sprache mit ähnlichem Abstraktionsgrad, nämlich JavaScript, umgewandelt [20]. Damit ist es möglich, TypeScript überall dort zu verwenden, wo JavaScript Code verlangt wird. Aus diesem Grund können beispielsweise auch jegliche Frameworks für JavaScript genauso auch mit TypeScript genutzt werden.

Eines dieser Frameworks, welches ursprünglich im Jahr 2010 für JavaScript entwickelt wurde, jedoch in der vorliegenden Arbeit gemeinsam mit TypeScript eingesetzt werden soll, ist Express.js³⁵. Das Framework, welches oft auch nur „Express“ genannt wird, benötigt die plattformunabhängige Open-Source Laufzeitumgebung Node.js³⁶. Express ist minimalistisch und bietet verschiedene Features wie beispielsweise Middleware und Routing an, um bei der Erstellung von mobilen Apps und Webanwendungen zu unterstützen [23, 36].

³⁴<https://www.typescriptlang.org/docs/>

³⁵<https://expressjs.com/>

³⁶<https://nodejs.org/en/>

4.3.2 Implementierung des RESTful Webservices

Isabell Waas

Nachfolgend wird der RESTful Webservice entsprechend mit der Sprache TypeScript und dem Framework Express.js umgesetzt.

Definition von Ressourcen und Data Transfer Objects

Auch die Entwicklung des Webservices mit TypeScript und Express beginnt mit dem Definieren von Ressourcen und DTOs. Diese werden dabei als Type Aliases entwickelt, welche flexibel sind, jedoch im Gegensatz zu Interfaces nicht nachträglich erweitert werden können, was sie leichter verständlich und sicherer macht. Im Folgenden ist der Typ Book abgebildet.

```

1 export type Book={
2   id :number
3   isbn :string
4   title :string
5   author:string
6 }
```

Listing 34: TypeScript-Code des Typs Book

Teilweise werden bei der Programmierung der DTOs verschiedene Utility Types von TypeScript eingesetzt, darunter Omit<Type, Keys>. Der erste Generic stellt dabei den Typ dar, auf dessen Basis ein neuer Typ erstellt werden soll. Durch den zweiten Generic wird in Form eines String Literal Types oder Union Types, bestehend aus mehreren String Literal Types, angegeben, welche Eigenschaften des Typs im ersten Generic bei dem neuen Typ entfernt werden sollen. Das unten stehende Listing zeigt ein entsprechendes DTO.

```

1 export type CreateBook=Omit<Book, "id">
```

Listing 35: TypeScript-Code des Typs CreateBook

Abgesehen von Omit<Type, Keys> wird der Utility Type Partial<Type> genutzt, der ebenfalls einen Typ als Basis erhält und dessen Eigenschaften bei dem neuen Typ alle optional macht.

Abruf eines Objekts

Bevor für die Ressource Book Endpunkte entwickelt werden können, soll zunächst das Abrufen, Erstellen, Aktualisieren und Löschen von Objekten aus der Datenbank unabhängig von der Kommunikation mit einem Clienten betrachtet werden. Hierfür bietet Express selbst kein ORM an und empfiehlt auch keines direkt in seiner Dokumentation, weshalb lediglich der MariaDB Node.js connector³⁷ eingesetzt wird, damit SQL-Querys getätigter werden können. Die Datenbankkommunikation wurde durch das folgende Interface abstrahiert.

```

1  export type DataAccessor<Item , Create , Update>={
2      /**
3      * @throws {UnknownError}
4      */
5      create (book :Create ) :Promise<number>
6
7      /**
8      * @throws {NotFoundError}
9      */
10     readById ( id :number ) :Promise<Item >
11
12     /**
13     * @throws {UnknownError}
14     */
15     read (page :number , perPage :number , query ?:string ) :Promise<[number , Item []]>
16
17     /**
18     * @throws {NotFoundError}
19     */
20     update ( id :number , element :Update ) :Promise<void >
21
22     /**
23     * @throws {NotFoundError}
24     */
25     delete ( id :number ) :Promise<void >
26 }
```

Listing 36: TypeScript-Code des Interfaces DataAccessor

³⁷<https://www.npmjs.com/package/mariadb>

Das selbst geschriebene Interface DataAccessor<Item, Create, Update> wird von der Klasse BooksService implementiert.

```

1  export class BooksService implements DataAccessor<Book, CreateBook, UpdateBook> {
2      private readonly pool:Pool
3
4      constructor(configuration:PoolConfig) {
5          this.pool = mariadb.createPool(configuration);
6      }
7
8      public async start():Promise<void> {
9          await this.pool.query('CREATE TABLE IF NOT EXISTS books (
10              id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
11              isbn TINYTEXT NOT NULL,
12              title TINYTEXT NOT NULL,
13              author TINYTEXT NOT NULL
14          );')
15      }
16
17      public async readById(id:number):Promise<Book> {
18          const result:unknown=await this.pool.query('SELECT * FROM books WHERE id=?;', [id])
19          if (!Array.isArray(result) || result.length==0) {
20              throw newNotFoundError("Book with id "+id+" not found.")
21          }
22          return result[0]
23      }
24      // weitere Methoden
25  }

```

Listing 37: TypeScript-Code des BooksService mit der Methode readById

Die Klasse hält einen Pool als Attribut, der eine Menge an Verbindungen zur Datenbank darstellt. Die für die Erstellung des Pools benötigte Konfiguration wird im Konstruktor übergeben. In der Methode start wird zunächst sichergestellt, dass die benötigte Tabelle erstellt wird, sollte sie noch nicht existieren. Zudem besitzt die Klasse für jede in diesem und den folgenden vier Abschnitten gestellten Anforderungen an den Webservice eine Methode, wel-

che jeweils eine Datenbankoperation ausführt. Alle Methoden des BooksServices sind asynchron. Es wird stets `async` und `await` genutzt sowie ein Promise-Objekt zurückgegeben.

Listing 37 zeigt den BooksService, wobei nicht alle Methoden für die Datenbankoperationen, sondern nur die Methode `readById` abgebildet ist, die in diesem Abschnitt beschrieben werden soll. Sie nutzt die Methode `query` des Pools, um per SQL ein Buch mit einer Id, die sie als Parameter erhält, abzufragen. Im Erfolgsfall gibt sie das erste Element in dem von `query` zurückgegebenen Array zurück, welches das abgerufene Buch ist. Tritt ein Fehler auf, wird eine Exception geworfen, was jedoch später erläutert wird.

Die Methoden des BooksService zur Kommunikation mit der Datenbank werden in der Klasse BooksController eingesetzt.

```
1 export class BooksController {
2
3     private readonly booksService:DataAccessor<Book, CreateBook, UpdateBook>
4     private readonly formValidator:FormValidator
5     private readonly contentValidator:ContentValidator
6
7     constructor(booksService:DataAccessor<Book, CreateBook, UpdateBook>,
8         formValidator:FormValidator, contentValidator:ContentValidator) {
9             this.booksService=booksService
10            this.formValidator=formValidator
11            this.contentValidator=contentValidator
12        }
13
14        public getBook:RequestHandler<{ id: string }, unknown, unknown, unknown, Record
15            <string, unknown>> = async (request, response):Promise<void> => {
16            let validationErrors:ValidationError []=[]
17            let errorMessages:string []=[]
18            if (!this.contentValidator.validateId(request.params.id, validationErrors)) {
19                validationErrors.forEach((validationError:ValidationError) => {
20                    errorMessages.push(validationError.parameterName+" "+validationError.
21                        errorMessage)
22                })
23                throw new BadRequestError(errorMessages)
24            }
25            const book:Book=await this.booksService.readById(parseInt(request.params.id))
26            response.json(book)
27        }
28    }
29}
```

```

22     response.status(200).json(book)
23   }
24   // weitere Methoden
25 }
```

Listing 38: TypeScript-Code des BooksControllers mit der Methode getBook

Neben einem Objekt des Typs BooksService – oder genauer, einem Typ, der das Interface DataAccessor<Book, CreateBook, UpdateBook> implementiert – hat der BooksController noch zwei weitere Attribute, die jedoch erst später in diesem Kapitel relevant werden. Alle Attribute werden im Konstruktor instanziert.

Der Codeausschnitt zeigt die Methode getBook. Diese ist ein asynchroner RequestHandler, welcher ein Request- und ein Response-Objekt als Parameter erhält. Intern findet zunächst eine Validierung der übergebenen Parameter statt. War diese fehlerfrei, so wird die bereits vorgestellte, ebenfalls asynchrone Methode readById des BooksService aufgerufen und das abgerufene Buch gemeinsam mit dem korrekten Statuscode in der Antwort zurückgegeben.

Um die Endpunkte schließlich aufrufen zu können, müssen die verschiedenen Routen angelegt werden. Dies geschieht in einer Klasse BooksRouter, welche intern eine Instanz der Klasse express.Router hält. Bei dieser handelt es sich um einen modularen Routenhandler, der Anfragen eines Clienten empfängt und je nach Route und HTTP-Verb die richtige Handlerfunktion ausführt. Letztere ist in der Regel eine Methode des BooksControllers, von welchem der BooksRouter in seinem Konstruktor eine Instanz erhält. Die jeweilige Handlerfunktion wird dabei einem sogenannten asyncCatchHandler übergeben, welcher eine Middleware darstellt, die erlaubt, dass in asynchronen Controller-Methoden geworfene Fehler an einen Errorhandler weitergegeben werden können. Wird ein HTTP-Verb für eine Route verwendet, für die kein Endpunkt mit diesem Verb existiert, wird über einen ErrorHandler eine Exception geworfen.

```

1 export class BooksRouter {
2   private readonly router:express.Router=express.Router()
3   private readonly booksController:BooksController
4
5   constructor(booksController:BooksController) {
```

```

6   this.booksController=booksController
7
8   this.router.route('/:id')
9     .get( asyncCatchHandler(this.booksController.getBook))
10    .patch(asyncCatchHandler(this.booksController.updateBook))
11    .delete(asyncCatchHandler(this.booksController.deleteBook))
12    .all(MethodNotAllowedHandler)
13
14  this.router.route('/')
15    .get(asyncCatchHandler(this.booksController.getAllBooks))
16    .post(asyncCatchHandler(this.booksController.createBook))
17    .all(MethodNotAllowedHandler)
18 }
19
20 public getRouter(): express.Router {
21   return this.router
22 }
23 }
```

Listing 39: TypeScript-Code des BooksRouters

Zuletzt muss der Router als Modul in die Express Application in der Datei index.ts, welche den Startpunkt der Anwendung darstellt, geladen werden. Dabei wird auch der Basis-Pfad „/v1/books“ für die Routen festgelegt. Dies ist in einem Codeausschnitt zu sehen, welcher im weiteren Verlauf dieses Kapitels abgebildet ist (siehe Listing 52).

Abruf aller Objekte mit Pagination, Sortierung und Filterung

Für den Endpunkt zum Abrufen aller Bücher mit Pagination, Sortierung und optionaler Filtermöglichkeit war neben dem Anlegen einer Route (siehe Listing 39) zunächst eine Methode im BooksService notwendig. In dieser wird eine SQL-Query zunächst in Form eines Textes aufgebaut, wodurch die WHERE-Bedingung optional angehängt werden kann, und dann mittels der Methode query ausgeführt. Über die JavaScript Methoden slice und ceil wird dann Pagination auf das Ergebnis angewandt. Die maximal mögliche Seite sowie die Bücher auf der gewünschten Seite werden als Erstes und zweites Element in einem Array zurückgegeben.

```

1  public async read(page:number, perPage:number, query?:string):Promise<[number,
2    Book[]]> {
3    let sqlQuery:string = 'SELECT * FROM books'
4    let values:string []=[]
5    if(query!==undefined) {
6      sqlQuery+=` WHERE isbn LIKE ? OR title LIKE ? OR author LIKE ?`
7      values.push(`%${query}%`, `%${query}%`, `%${query}%`)
8    }
9
10   const result:unknown=await this.pool.query(sqlQuery+' ORDER BY id asc;', values)
11
12   if (!Array.isArray(result)) {
13     throw new UnknownError("An unknown error occurred.")
14   }
15
16   let booksOnNthPage:Book[]=result.slice((page-1)*perPage, page*perPage)
17   let maximumPage:number=Math.ceil(result.length/perPage)
18
19   return [maximumPage, booksOnNthPage]
20 }
```

Listing 40: TypeScript-Code der Service-Methode read

Die Methode read des BooksService wird schließlich in der Methode getAllBooks des Books-Controllers verwendet. Der Service-Methode werden die benötigten Parameter übergeben, die aus dem Attribut query des Request-Objekts gelesen und validiert werden. Die erst später in diesem Kapitel relevante Validierung wird in diesem und den folgenden Listings, die Methoden des BooksControllers darstellen, nicht mit abgebildet. Sie läuft jedoch stets ähnlich ab wie in Listing 38.

```

1  public getAllBooks:RequestHandler<unknown, unknown, unknown, unknown, Record<
2    string, unknown>> = async (request, response):Promise<void> => {
3    // Validierung der Query-Parameter
4    const reading:[number, Book[]]=await this.booksService.read(parseInt(request.
5      query.page), parseInt(request.query.per_page), request.query.query!=
6      undefined ? request.query.query : undefined)
```

```

4   response.status(200).json({maximumPage: reading[0], items: reading[1]}
5     satisfies BooksResponse)
}

```

Listing 41: TypeScript-Code der Controller-Methode getAllBooks

Erstellung eines Objekts

Die Methode create des BooksService zum Erstellen eines Buches empfängt ein Objekt des Typs CreateBook und übergibt dessen Eigenschaften innerhalb einer INSERT-Anweisung per SQL an die Datenbank. Ihre Rückgabe enthält die Id des neuen Buches, welche in dem von der Methode query zurückgegebenen JSON-Objekt enthalten ist.

```

1  public async create(book:CreateBook):Promise<number> {
2    const insertResult:any=await this.pool.query('INSERT INTO books (isbn, title,
3      author) VALUES (?, ?, ?);', [book.isbn, book.title, book.author]);
4    if (!("affectedRows" in insertResult && insertResult.affectedRows==1)) {
5      throw new UnknownError("An unknown error occurred.")
6    }
7    return parseInt(insertResult.insertId)
}

```

Listing 42: TypeScript-Code der Service-Methode create

Ähnlich wie bei den vorherigen Endpunkten wird die Methode von einer Controller-Methode aufgerufen und erhält den validierten Body des Request-Objektes als Parameter.

```

1  public createBook:RequestHandler<unknown, unknown, unknown, unknown, Record<
2    string, unknown>> = async (request, response):Promise<void> => {
3    // Validierung des Request Bodys
4    const bookId:number=await this.booksService.create(request.body)
5    response.status(201).json({id: bookId} satisfies IdResponse)
}

```

Listing 43: TypeScript-Code der Controller-Methode createBook

Änderung einzelner Attribute eines Objekts

Für das Modifizieren bestimmter Eigenschaften eines Buches wird wie beim Abrufen aller Bücher eine SQL-Query in Textform aufgebaut, sodass je nach übermittelter Daten nicht alle Attribute des Buches verändert werden. Wenn die Datenoperation über die query-Methode erfolgreich ausgeführt wurde, ist die Methode fertig und es wird nichts zurückgegeben.

```

1  public async update(id:number, book:UpdateBook):Promise<void> {
2    let sqlQuery:string='UPDATE books SET '
3    let values:string []=[]
4    if(book.isbn!==undefined) {
5      sqlQuery+=`isbn=?`
6      values.push(book.isbn)
7    }
8    if(book.title!==undefined) {
9      if(values.length>0) {
10        sqlQuery+=", "
11      }
12      sqlQuery+=`title=?`
13      values.push(book.title)
14    }
15    if(book.author!==undefined) {
16      if(values.length>0) {
17        sqlQuery+=", "
18      }
19      sqlQuery+=`author=?`
20      values.push(book.author)
21    }
22    const result=await this.pool.query(sqlQuery+` WHERE id=?;`, [...values, id])
23    if(!("affectedRows" in result && result.affectedRows==1)) {
24      throw new NotFoundError(`Book with id ${id} not found.`)
25    }
26  }

```

Listing 44: TypeScript-Code der Service-Methode update

Auch diese Methode wird in einer Methode des BooksControllers verwendet. Sie erhält die aus dem Attribut params des Requests erhaltene Id sowie den validierten Body.

```

1 public updateBook:RequestHandler<{ id: string }, unknown, unknown, unknown,
2   Record<string, unknown>> = async (request, response):Promise<void> => {
3   // Validierung des Parameters Id und des Request Bodys
4   await this.booksService.update(parseInt(request.params.id), request.body)
5   response.status(204).send()
}

```

Listing 45: TypeScript-Code der Controller-Methode updateBook

Löschen eines Objekts

Zum Schluss soll eine Methode zum Löschen eines Buches im BooksService implementiert werden. Diese führt eine SQL-Anfrage aus und gibt im Erfolgsfall nichts zurück.

```

1 public async delete(id:number):Promise<void> {
2   const result=await this.pool.query('DELETE FROM books WHERE id=?;', [id])
3   if (!("affectedRows" in result && result.affectedRows==1)) {
4     throw new NotFoundError("Book with id "+id+" not found.")
5   }
6 }

```

Listing 46: TypeScript-Code der Service-Methode delete

Die delete-Methode des BooksService wird von der Methode deleteBook des Controllers ausgeführt und übergibt ihr die validierte Id des zu löschenen Buches.

```

1 public deleteBook:RequestHandler<{ id: string }, unknown, unknown, unknown,
2   Record<string, unknown>> = async (request, response):Promise<void> => {
3   // Validierung des Parameters Id
4   await this.booksService.delete(parseInt(request.params.id))
5   response.status(204).send()
}

```

Listing 47: TypeScript-Code der Controller-Methode deleteBook

Validierung von erhaltenen Daten

Für die Validierung sind die Klassen FormValidator und ContentValidator zuständig. Erstere bietet Methoden, um zunächst den Typ des zu prüfenden Wertes selbst zu kontrollieren sowie, wenn es sich um einen komplexen Typ handelt, ob alle benötigten Attribute enthalten sind und auch den richtigen Typ haben. Das unten stehende Listing zeigt eine der Methoden. Fehler werden dabei als Objekte des selbst definierten Typs ValidationError in einem Array gespeichert. Der Rückgabewert der Methode ist ein Type Predicate, was ein Type Guard ist, der garantiert, dass der zu prüfende Wert den gewünschten Typ besitzt oder eben nicht.

```

1  public isUpdateBook(value:unknown, validationErrors:ValidationError[]):value is
2      UpdateBook {
3          if (!this.isNonNullObject(value, validationErrors)) {
4              return false
5          }
6          let startValidationErrorsLength:number=validationErrors.length
7          if ("isbn" in value && (typeof value.isbn!="string")) {
8              validationErrors.push({parameterName:"isbn", errorMessage:"is not a string in
9                  the correct format."})
10         }
11         if ("title" in value && (typeof value.title!="string")) {
12             validationErrors.push({parameterName:"title", errorMessage:"is not a string
13                 with at least one character."})
14         }
15         if ("author" in value && (typeof value.author!="string")) {
16             validationErrors.push({parameterName:"author", errorMessage:"is not a string
                  with at least one character."})
17         }
18     }
19     return validationErrors.length==startValidationErrorsLength
20 }
```

Listing 48: TypeScript-Code der FormValidator-Methode isUpdateBook

Die Klasse ContentValidator stellt anschließend sicher, dass der konkrete Inhalt der Attribute eines komplexen Typs valide ist. Beispielsweise wird für die ISBN des Buches geprüft, ob eine Regular Expression eingehalten ist. Aufgetretene Fehler werden wieder in einem Array mit ValidationError-Objekten gespeichert. Zurückgegeben wird ein Wahrheitswert.

```

1  public validateUpdateBook(updateBook : UpdateBook , validationErrors : ValidationErrors
2      [] ) : boolean {
3      let startValidationErrorsLength : number = validationErrors . length
4      if ( updateBook . isbn !== undefined ) {
5          this . validateIsbn( updateBook . isbn , validationErrors )
6      }
7      if ( updateBook . title !== undefined ) {
8          this . validateTextLength( updateBook . title , " title " , 1 , 300 , validationErrors )
9      }
10     if ( updateBook . author !== undefined ) {
11         this . validateTextLength( updateBook . author , " author " , 1 , 100 , validationErrors )
12     }
13 }
14
15 private validateIsbn(isbn : string , validationErrors : ValidationErrors [] ) : boolean {
16     if ( !(/^ (978 - ?|979 - ?) ? \d{1,5} - ? \d{1,7} - ? \d{1,6} - ? \d{1,3} $/. test (isbn))) {
17         validationErrors . push ({ parameterName : " isbn " , errorMessage : " is not in the
18             correct format ." })
19         return false
20     }
21     return true
}

```

Listing 49: TypeScript-Code zweier ContentValidator-Methoden

Behandlung von Fehlern

In den Listings wurde bereits gezeigt, dass an verschiedenen Stellen in dem Webservice Exceptions geworfen werden. Diese werden durch eine globale Middlewarefunktion behandelt, dem ErrorHandler. Er reagiert auf einige eigens definierte Errorklassen, welche stets von Error erben, und gibt passende Statuscodes und Fehlermeldungen zurück. Auf unbekannte Fehler antwortet er mit einer allgemeinen Fehlernachricht. In der im nächsten Abschnitt abgebildeten Datei index.ts wird der ErrorHandler in der Anwendung registriert.

```

1  export const ErrorHandler = (error: unknown, request: Request<object, unknown,
2    unknown, unknown>, response: Response<unknown>, next: NextFunction) => {
3    if(error instanceof BadRequestError) {
4      response.status(400).json({messages: error.messages} satisfies ErrorResponse
5    )
6    } else if(error instanceof NotFoundError) {
7      response.status(404).json({message: error.message} satisfies ErrorResponse)
8    } else if(error instanceof MethodNotAllowedError) {
9      response.status(405).json({message: error.message} satisfies ErrorResponse)
10   } else if(error instanceof UnknownError) {
11     response.status(500).json({message: error.message} satisfies ErrorResponse)
12   } else if(error instanceof SyntaxError) {
13     response.status(400).json({message: "The JSON is invalid."} satisfies
14     ErrorResponse)
15   }

```

Listing 50: TypeScript-Code des ErrorHandlers

An dieser Stelle soll kurz auf den in dem Codeausschnitt genutzten `satisfies` Operator von TypeScript hingewiesen werden. Dieses noch sehr neue Feature prüft, ob ein Typ alle Eigenschaften eines anderen Typs erfüllt und verhindert so Fehler.

Konfiguration des Servers

Als Letztes soll der Code zur Konfiguration des Servers beleuchtet werden. Zum einen ist hierbei die Datei `package.json` zu nennen. Jedes Node.js Projekt besitzt eine solche Datei, in welcher Metadaten definiert werden. Unter diesen sind insbesondere der Name der Anwendung sowie deren Einstiegspunkt, ausführbare Skripte und Abhängigkeiten.

```

1  {
2      "name": "expressjs_libraryapi",
3      "version": "1.0.0",
4      "description": "Example REST api for a library written in TypeScript with Node.
5          js.",
6      "main": "index.js",
7      "scripts": {
8          "start": "npm run build && node ./dist/index.js",
9          "build": "npx tsc"
10     },
11     "author": "",
12     "type": "module",
13     "license": "ISC",
14     "dependencies": {
15         "express": "4.18.2",
16         "mariadb": "3.0.2",
17         "typescript": "^4.9.4",
18         "@types/express": "^4.17.16"
19     }
}

```

Listing 51: Datei package.json für das Node-Projekt

Alles Weitere wird in dem eben festgelegten Einstiegspunkt, der Datei index.js beziehungsweise vor dem Transpilieren index.ts, konfiguriert. Genauer wird hier die Express-Applikation erzeugt, alle benötigten Instanzen wie zum Beispiel der BooksService, der insbesondere auch die Konfigurationsattribute für die Datenbank benötigt und dessen bereits angesprochene Methode start aufgerufen werden muss, erstellt sowie alle Middlewares in die Applikation eingebunden. Zuletzt wird die Applikation durch die Methode listen auf einem bestimmten Port gestartet.

```

1 const app: Application=express()
2
3 const booksService : BooksService=new BooksService({
4     host: 'localhost',
5     user: 'libraryapi',

```

```

6  password: "1234",
7  database: 'libraryapi',
8  port : 24638,
9  connectionLimit: 150
10 })
11 await booksService.start()
12 const formValidator:FormValidator=new FormValidator()
13 const contentValidator:ContentValidator=new ContentValidator()
14 const booksController:BooksController=new BooksController(booksService,
   formValidator, contentValidator)
15 const booksRouter:BooksRouter=new BooksRouter(booksController)
16
17 app.use(express.json())
18
19 app.use('/v1/books', booksRouter.getRouter())
20
21 app.use(ResourceNotFoundHandler)
22
23 app.use(ErrorHandler)
24
25 app.listen(21003, () => {
26   console.log('Application started on port 21003.')
27 })

```

Listing 52: TypeScript-Code der Datei index.ts

Schließlich soll noch erwähnt werden, dass für die Nutzung von TypeScript eine Datei namens tsconfig.json benötigt wird. Durch diese werden sowohl die root Dateien festgelegt als auch die Compiler Optionen für das Kompilieren des Projektes.

4.3.3 Bewertung

Isabell Waas

Mit dem Abschluss der Entwicklung des Webservices mit TypeScript und Express.js, kann mit dessen Beurteilung begonnen werden.

Dokumentation und Wartung

Der Einstieg in Express wird auf der Webseite³⁸ des Frameworks verhältnismäßig einfach gestaltet. Denn unter „Getting started“ in der Hauptnavigation sind alle wichtigen Informationen komprimiert und mit Codebeispielen zu finden, mit denen sich bereits in wenigen Minuten eine sehr einfache Anwendung erstellen lässt. Ebenso sind dort zahlreiche kleine Beispielanwendungen verlinkt, die verschiedene Anwendungsfälle von Express veranschaulichen. Um tiefer in einzelne Themenbereiche, darunter Routing und Middleware einzusteigen, stellt die Webseite durch den Menüpunkt „Guide“ pro Thema eine ausführlichere, übersichtliche Webseite bereit. Diese Seiten beinhalten ebenfalls stets auch Beispielcode.

Die folgende Abbildung zeigt einen der Guides.

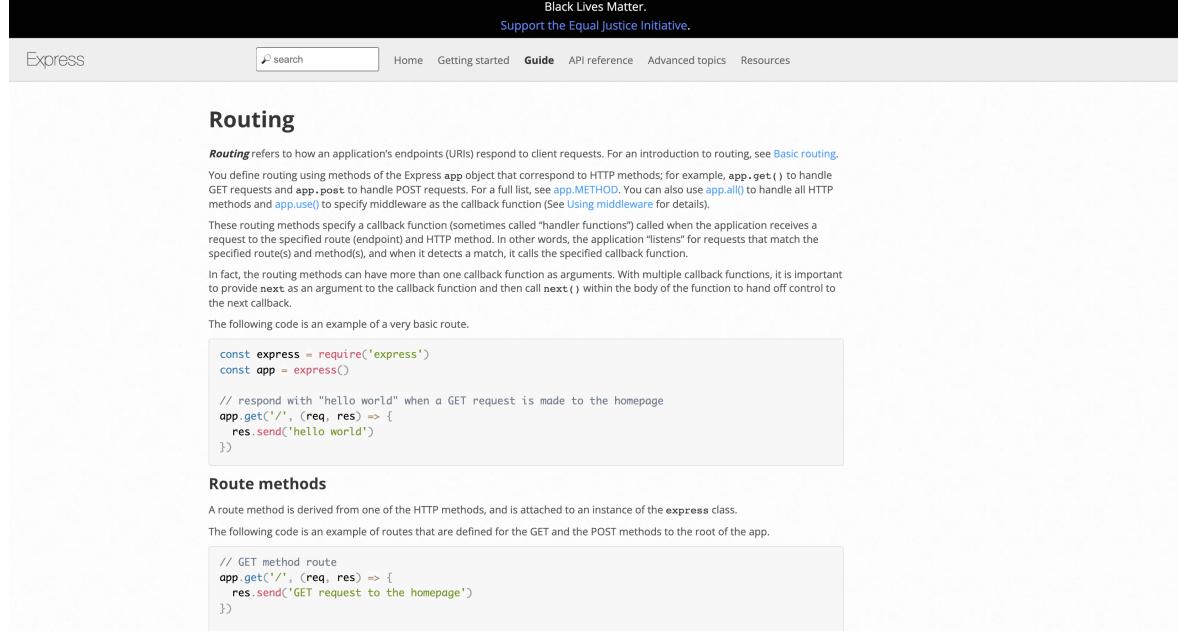


Abbildung 36: Guide zum Thema Routing in Express [23]

³⁸<https://expressjs.com/>

Speziellere, für diese Arbeit weniger relevante Themen sind unter „Advanced topics“ aufgeführt. Darüber hinaus kann sich unter dem Menüpunkt „API reference“ über die verschiedenen Klassen, Methoden und Properties, die Express mit sich bringt, näher informiert werden.

The screenshot shows the Express.js API Reference page for version 4.x. At the top, there's a navigation bar with links for Home, Getting started, Guide, API reference (which is the active tab), Advanced topics, and Resources. A banner at the top right reads "Black Lives Matter. Support the Equal Justice Initiative." On the left, there's a sidebar with a tree view of the API structure:

- express()**
 - Methods**
 - express.json()
 - express.raw()
 - express.Router()
 - express.static()
 - express.text()
 - express.urlencoded()
- Application**
- Request**
- Response**
- Router**

The main content area starts with the **express()** function documentation. It includes a description: "Creates an Express application. The `express()` function is a top-level function exported by the `express` module.", a code example in a code block, and a "Methods" section. Under "Methods", there's a detailed description of the `express.json([options])` middleware, including notes about its availability in v4.16.0, its purpose (parsing JSON payloads), and its behavior with respect to the `Content-Type` header and `req`.body. Below this, a table describes the optional `options` object.

Property	Description	Type	Default

Abbildung 37: API Reference von Express 4.x [23]

Zuletzt sollte auch der Menüpunkt „Resources“ nicht unerwähnt bleiben. Dort gibt es insbesondere ein Glossar sowie eine Seite mit Verlinkungen auf externes Lernmaterial. Darunter befinden sich viele Bücher und Blogs. Videotutorials hingegen sind nur wenige aufgeführt. Alles in allem ist die Dokumentation dafür, dass Express für einige Themenbereiche wie beispielsweise Validierung von Daten oder Object Relational Mapping keine Funktionalität bietet, recht umfassend. Dennoch verweist sie leider sehr häufig auf externe Ressourcen, statt eigenes Material anzubieten. Entwickler, die gerne mit Videos lernen, werden ebenfalls nur in geringem Maße fündig.

Zur Wartung des Frameworks kann gesagt werden, dass es relativ häufig aktualisiert wird. Etwa drei bis vier Mal im Jahr wird eine neue Version herausgegeben, darunter jedoch auch kleinere Updates sowie Alpha- und Beta-Versionen eines neuen, größeren Releases. Große Updates, z. B. von Version 4.17.0 auf 4.18.0, erfolgen nur etwa alle ein bis drei Jahre.

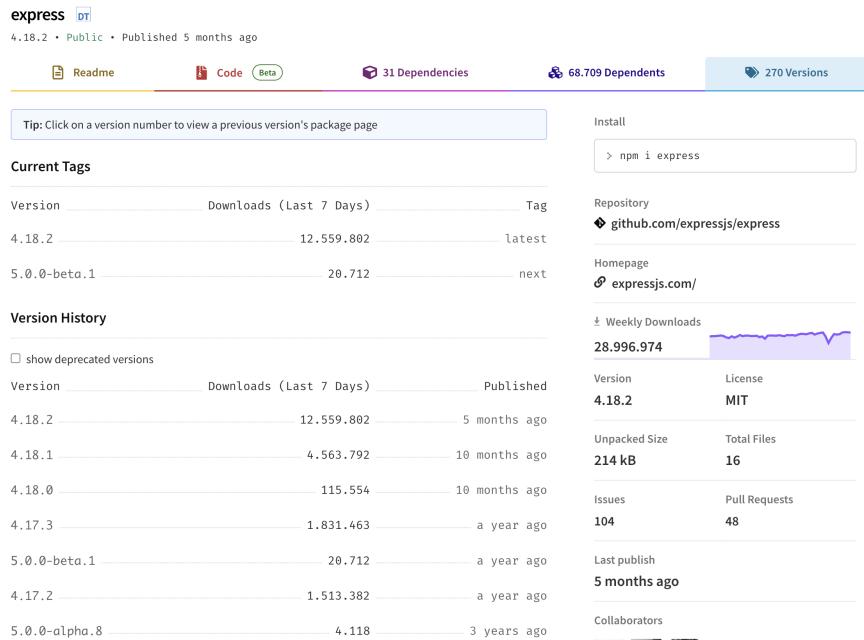


Abbildung 38: Releases von Express [27]

Lines of Code

Auch für die Implementierung des Webservices mit Express sollen die Lines of Code mithilfe von cloc berechnet werden. Dabei werden nur TypeScript-Dateien berücksichtigt, da nur diese eigentlichen Programmcode beinhalten. Die weiteren Dateien dienen vor allem zur Konfiguration. Die ermittelten Codezeilen sind in der unten stehenden Tabelle dargestellt.

Language	files	blank	comment	code
TypeScript	24	63	15	449

Tabelle 5: Ausgabe des Programms cloc bei der Implementierung mit TypeScript und Express

Notwendigkeit zusätzlicher Bibliotheken

Der Webservice setzt in erster Linie auf das Framework Express. Für dieses müssen die Typen über den Node-Package-Manager (npm) installiert werden, damit die Vorteile von TypeScript genutzt werden können. Abgesehen davon wird auch der MariaDB Node.js connector als Abhängigkeit in der Datei package.json (siehe Listing 51) eingebunden, welcher dafür sorgt, dass die Node.js-Applikation mit MariaDB und MySQL Datenbanken kommunizieren kann. Während in der Dokumentation von Express die benötigten Module für einige Datenbanksysteme vorgestellt werden, wird für alle weiteren Optionen, darunter auch MariaDB, auf die npm-Webseite verwiesen [23]. Daher kann gesagt werden, dass die primitivste externe Bibliothek genutzt wurde, um die Datenbankanbindung zu realisieren.

Auch wenn Express theoretisch in Verbindung mit einem ORM-Tool genutzt werden könnte, wird in dem Webservice im Rahmen dieser Arbeit keines eingesetzt. Das ist dadurch begründet, dass Express weder eines mitliefert noch in seiner Dokumentation direkt eines empfiehlt. Dies hat unter anderem zur Folge, dass Datenbankoperationen mit reinem SQL geschrieben werden müssen, was aufwendiger ist, als wenn vorgefertigte Methoden eines ORMs verwendet werden könnten.

Was die Validierung betrifft, so bietet Express keine Bibliothek hierfür an oder verweist in seiner Dokumentation auf eine externe Bibliothek. Daher erfolgt auch diese vollständig händisch mit der Funktionalität, die TypeScript selbst bietet.

Wichtige Architekturprinzipien und besondere Sprachfunktionen

Anschließend werden nun die Architektur und die wichtigsten verwendeten Sprachfunktion von TypeScript erläutert. Dazu zeigt die folgende Grafik den Projektaufbau, wobei nur Ordner mit TypeScript-Dateien und die wichtigsten Dateien gezeigt werden.

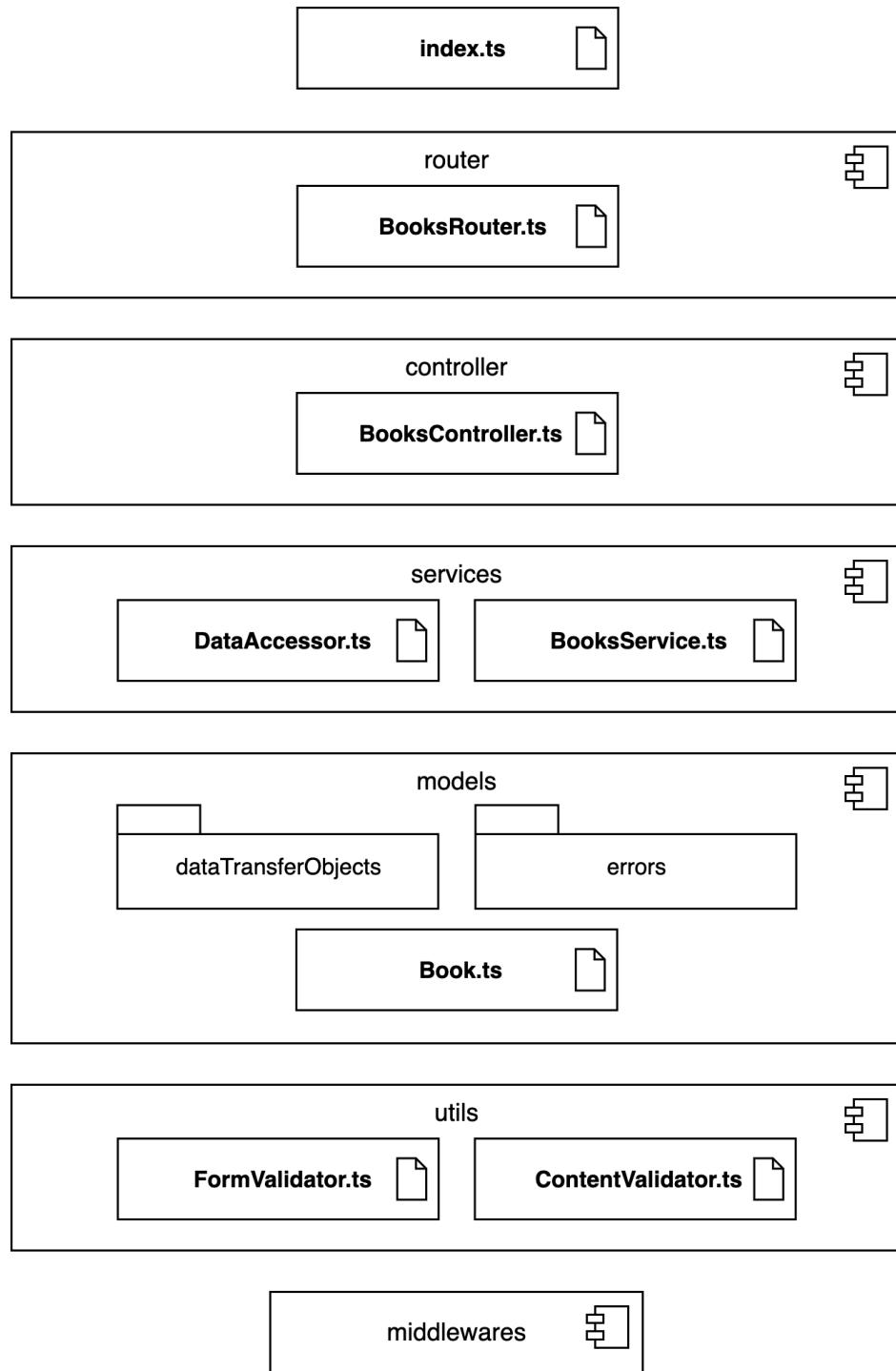


Abbildung 39: Architektur des Webservices mit Express

Zu Bewerten ist vor allem, inwieweit eigenständige Einheiten implementiert werden und diese das Single-Responsibility-Prinzip einhalten. Beides wurde in dem Webservice umgesetzt, jedoch nicht aufgrund von Vorgaben durch Express. Genauer wird dies nun anhand der Einheiten in Abbildung 39 erläutert.

Den Startpunkt der Anwendung stellt die Datei index.ts dar. Dort wird die Express-Applikation aufgebaut und es werden sämtliche Instanzen benötigter Klassen erstellt, konfiguriert und einander übergeben. Beispielsweise erhält der Service, der später näher beschrieben wird, die Datenbankkonfigurationsattribute. Es werden in der Datei index.ts zudem verschiedene Middlewares in die Applikation eingebunden und schließlich wird der Service gestartet. All dies muss manuell durch den Programmierer geschehen, denn Express bietet keine Dependency Injection oder andere Möglichkeiten zur automatischen Konfiguration. Grundsätzlich ist es gut, wenn die Konfiguration in einer einzigen zentralen Datei geschieht und bei Änderungen stets nur diese angepasst werden muss. Jedoch liegt es in Entwicklerhand, ob dies der Fall ist. Außerdem ist es sehr negativ, dass das Express Application-Objekt in dieser Datei ein Gottobjekt ist.

Ein Router ist eine Klasse, die Routen und deren HTTP-Verben definiert sowie festlegt, welcher Handler für jeden dieser Endpunkte ausgeführt wird. Die meisten Handler verwenden dabei eine Methode eines Controllers, den der Router über seinen Konstruktor erhält. Auf den Controller wird später genauer eingegangen. Schließlich wird der Router wie eine Middleware als Modul in der Applikation registriert, wobei angegeben wird, für welchen Hauptpfad, zum Beispiel „/v1/books“, er zuständig ist. Es ist möglich, in der Datei index.ts mehrere Router auf diese Weise einzubinden. Ebenso können bestehende Router sehr leicht um weitere Endpunkte ergänzt werden, insofern der im Folgenden angesprochene Controller ebenfalls erweitert wird.

Der Controller definiert Methoden, die RequestHandler darstellen und jeweils bei Endpunkten aufgerufen werden. Er nutzt einen Service, der später erläutert wird, sowie zwei Klassen zur Validierung von Daten. Die Instanzen dieser benötigten Klassen erhält er über seinen Konstruktor. Zur Validierung soll noch gesagt werden, dass Express hierfür keine eigenen Möglichkeiten bietet und diese daher händisch mit der Funktionalität von TypeScript selbst vonstattengeht. Glücklicherweise bietet zumindest TypeScript Features wie die im Laufe der

Arbeit erwähnten Type Predicates, die diesen Vorgang etwas erleichtern. Zudem soll gesagt werden, dass die Kapselung durch einen Controller sowie dessen Methoden ebenfalls nicht durch Express vorgegeben ist, sondern von dem Programmierer selbst umgesetzt werden muss.

Um den Datenbankzugriff kümmert sich ein selbst implementierter Service. Zur besseren Austauschbarkeit sowie um es zu erleichtern, weitere Services hinzuzufügen, wurde ein generisches Interface `DataAccessor<Item, Create, Update>` verfasst, das jeder Service implementieren muss. Es beinhaltet abstrakte Methoden für die gängigsten Datenbankoperationen, welche auch in den Webservices in dieser Arbeit verlangt waren. Diese Methoden werden in der konkreten Service-Klasse überschrieben und implementiert, wobei reine SQL-Anweisungen verfasst werden müssen, da Express keine Möglichkeiten bereitstellt, um Datenbankabfragen einfacher zu gestalten. Die Datenbankkonfiguration erhält der Service zudem in seinem Konstruktor und in einer Methode `start` stellt er ebenfalls mit reinem SQL sicher, dass von ihm benötigte Tabellen, wenn nötig erstellt werden.

Die Models werden bis auf die Fehlerklassen, die von `Error` erben müssen, als Type Aliases geschrieben. Dies ist ein Feature von TypeScript, welches bereits in Kapitel 4.3.2 erläutert wurde. Ebenso wurden dort zwei Utility Types vorgestellt, über die sich der Code wesentlich kürzer und redundanzfreier gestalten lässt. Unterschiede und Gemeinsamkeiten zwischen verschiedenen DTOs werden ebenfalls durch dieses Sprachfeature deutlich.

Middlewares stellen das Kernfeature von Express dar. Sie sind individuell kombinierbar und bieten dadurch Möglichkeiten zur Kapselung. Zudem ist es jederzeit möglich, weitere dieser Weiterleitungsfunktionen in die Anwendung einzubauen. Neben den bereits erwähnten gibt es noch ein paar Middlewares, die zur Fehlerbehandlung dienen, allen voran der `ErrorHandler`, der für verschiedene Fehlerarten andere Response-Objekte zurückgibt und sehr einfach um weitere Fehlerarten erweitert werden kann. Hierbei wird der bereits beschriebene `satisfies` Operator von TypeScript genutzt.

Alles in allem gibt Express bei der Architektur sehr wenig vor, weshalb es in der Hand des Entwicklers liegt, wie gut diese hinsichtlich Aspekten wie Wartbarkeit und Erweiterbarkeit ist. Das kann zunächst nicht vollends schlecht klingen, jedoch wäre es nach der Definition

des Bewertungskriteriums wesentlich besser, wenn bei einigen Aufgaben, die Teil der Entwicklung eines RESTful Webservices sind, das Framework mehr Hilfestellung geben würde. Insbesondere die Validierung, die Konfiguration, die Initialisierung und Übergabe von Abhängigkeiten sowie das Schreiben von Datenbankabfragen müssen manuell geschehen und sind sehr aufwendig. Besondere Sprachfeatures von TypeScript werden vor allem im ErrorHandler und bei der Implementierung der Models eingesetzt, wo einige Stärken des Typensystems der Sprache deutlich werden.

Performance

Als Nächstes soll noch das Kriterium Performance beleuchtet werden. Hierfür wurde der Webservice dem Artillery-Test unterzogen und es wurde ein HTML-Report erstellt. Diesem wurden die folgenden beiden Diagramme entnommen.

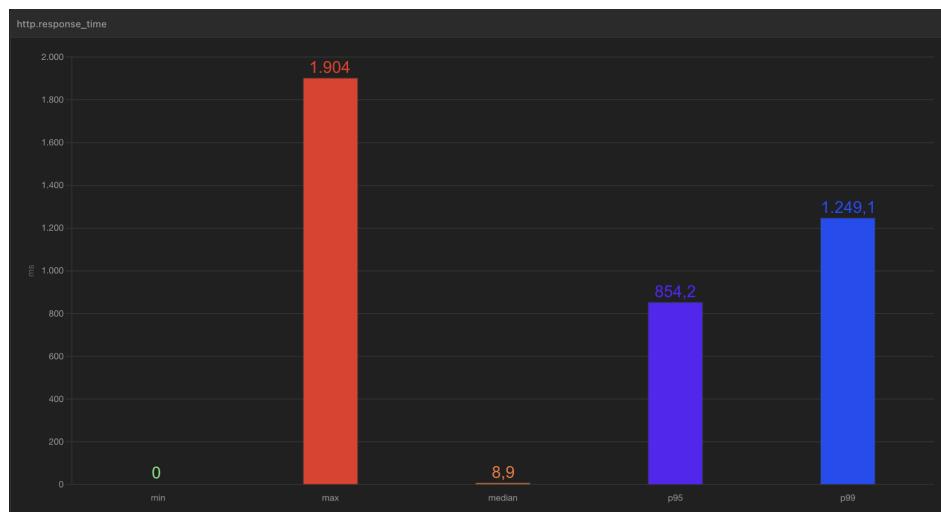


Abbildung 40: Ausgewählte Antwortzeiten für TypeScript mit Express (Balkenwerte manuell hinzugefügt)

Die erste Grafik zeigt bestimmte Antwortzeiten des Webservices an, darunter wieder die kürzeste und längste sowie die maximalen Antwortdauern für 50%, 95% und 99% der Anfragen. In der zweiten Abbildung sind dagegen ausgewählte Sessionlängen dargestellt, wobei dieselben Säulenarten genutzt werden.

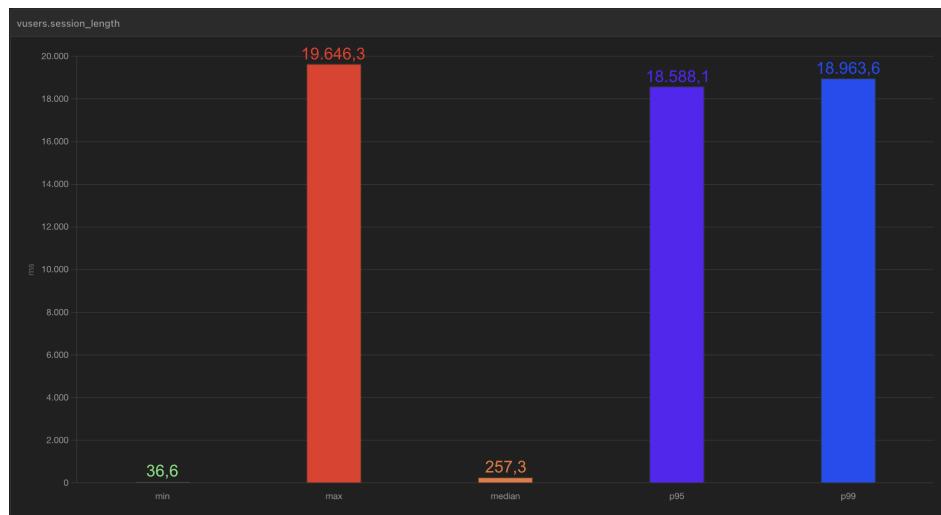


Abbildung 41: Ausgewählte Sessionlängen für TypeScript mit Express (Balkenwerte manuell hinzugefügt)

4.4 Go mit Gin

4.4.1 Einführung in die Sprache und das Framework

Steven Solleder



Abbildung 42: Go Logo [33]



Abbildung 43: Gin Logo [25]

Wie nicht wenige andere Technologien, die von den meisten Menschen heutzutage im Alltag eingesetzt werden, stammt auch die Programmiersprache Go³⁹, häufig auch Golang genannt, von dem bekannten Unternehmen Google. Sie wurde zunächst als internes Projekt von den drei Entwicklern Robert Griesemer, Rob Pike und Ken Thompson, der unter anderem durch seine Beteiligung an der Erfindung des Unix-Betriebssystems in der Informatik sehr bekannt ist, begonnen. Aus dem Vorhaben wurde schließlich ein offizielles Open-Source Projekt, dessen erste Version 2012 veröffentlicht wurde. Go orientiert sich in seiner Syntax an C, ist jedoch wesentlich einfacher und leichtgewichtiger. Eines der Hauptziele von Go ist zudem, dass sie die Programmierung eines gleichermaßen effizienten, schnellen und stabilen Codes ermöglicht. Dies erreicht die Sprache unter anderem durch eine umfassende Standardbibliothek, automatische Speicherbereinigung sowie Plattformunabhängigkeit [44].

Gemeinsam mit Go soll das in Go selbst verfasste Web Framework Gin⁴⁰, dessen erste Version 2015 veröffentlicht wurde, zum Einsatz kommen. Gins Hauptmerkmale sind seine hohe Performance sowie die Unterstützung von Middleware und Routing [24, 25].

³⁹<https://go.dev/doc/>

⁴⁰<https://github.com/gin-gonic/gin>

4.4.2 Implementierung des RESTful Webservices

Steven Solleder

Abschließend wird der RESTful Webservice nun mit Go und Gin implementiert.

Definition von Ressourcen und Data Transfer Objects

Zu aller erst müssen die Model- und DTO-Structs erstellt werden. Bevor mit der eigentlichen Erläuterung begonnen wird, benötigt es noch ein paar wesentliche Informationen über die teilweise äußerst unübliche Syntax bei komplexen Datentypen in Go.

Da sich die Programmiersprache Go stark an C orientiert, gibt es neben Funktionen und Interfaces nur Structs. Dabei besteht in Go kein Bedarf, sich zwischen einem separaten Referenztypen und einem separaten Werttypen festzulegen. Bei Instanzen auf Basis eines Struct-Typs ist es jederzeit möglich, zu entscheiden, ob die Speicheradresse der Instanz oder eine Kopie der Instanz zugewiesen beziehungsweise übergeben werden soll. Die Syntax folgt dabei der von C. Primitive Werte werden hingegen bei einer Zuweisung oder Übergabe stets kopiert. Im Gegensatz zu C ist zudem keine Pointer-Arithmetik möglich. Darüber hinaus können den Attributen eines Structs Metainformationen in Form eines Strings angehängt werden. Auch ist es möglich, Structs nach ihrer Deklaration Methoden hinzuzufügen. Dabei muss eine Funktion definiert werden, welche vor ihrem Namen einen Empfängertyp festlegt. Hierbei kann festgelegt werden, ob die Funktion stets auf einer Kopie oder der Referenz des Objekts ausgeführt wird. Auch wenn es streng genommen Funktionen sind, werden diese Art von Funktionen in folgenden Methoden genannt. Generell entscheidet der erste Buchstabe des Namens einer Funktion, Variable, eines Structs oder Attributs, ob sie beziehungsweise es außerhalb des Packages sichtbar ist. Bei einem Großbuchstaben ist es von außen sichtbar, bei einem Kleinbuchstaben nur innerhalb. Auch zu erwähnen ist, dass, wenn der Typ einer Variable oder eines Attributs ein Zeigertyp ist, in jenem nil gespeichert werden kann. Ansonsten ist dies nicht möglich. Stattdessen muss der Zerovalue gespeichert werden. Bei int ist der Zerovalue 0, bei string ist es ein leerer string und bei bool ist es false. Dies kann zu Problemen führen, wenn herausgefunden werden soll, ob eine Variable oder ein Attribut einen Wert zugewiesen bekommen hat, da Zerovalues für sich auch sinnvolle Werte sein können.

Zur Verdeutlichung eines Models wird nun der Struct CreateBook gezeigt.

```

1 type CreateBook struct {
2   ISBN    string `json:"isbn" binding:"required,isbn"`
3   Title   string `json:"title" binding:"required,min=1,max=300"`
4   Author  string `json:"author" binding:"required,min=1,max=300"`
5 }
```

Listing 53: Go-Code des DTOs CreateBook

Der größte Teil des abgebildeten Codes ist nach den vorherigen Erläuterungen selbsterklärend. Dennoch muss erwähnt werden, dass mithilfe der Metainformationen verschiedene zusätzliche Aspekte definiert werden. Zum einen wird festgelegt, wie das Attribut bei der Umwandlung in oder aus JSON heißen soll. Auch wird nach dem Wort „binding“ angegeben, welche Bedingungen das jeweilige Attribut bei der Validierung erfüllen muss. Die Anforderungen werden dabei durch Kommata getrennt. Ist ein Attribut nicht zwingend notwendig, so beinhaltet es, insofern kein Wert geschickt wurde, den entsprechenden Zerovalue.

Abruf eines Objekts

Gin bietet selbst kein ORM oder ein ähnliches Werkzeug zum Abrufen und Persistieren von Daten. Um dennoch eine Verbindung zu der Datenbank aufzubauen, wird das von Go mitgelieferte, primitive Package database/sql in Kombination mit dem Go SQL Treiber für MySQL⁴¹ verwendet, der auch mit MariaDB funktioniert. Dafür wird ein eigener Datenbank-Service geschrieben, welche das folgende Interface zur Abstraktion und Modularisierung implementiert.

```

1 type DataAccessor[Item any, Create any, Update any] interface {
2   Create(element Create) (uint, error)
3   Read(page uint, perPage uint, query string) (uint, []*Item, error)
4   ReadById(id uint) (*Item, error)
5   Update(id uint, element Update) (bool, error)
6   Delete(id uint) (bool, error)
7 }
```

Listing 54: Go-Code des Interfaces DataAccessor

⁴¹<https://github.com/go-sql-driver/mysql>

In den eckigen Klammern hinter dem Klassennamen werden Generics definiert. Außerdem gibt es in Go keinen Mechanismus zum Werfen von Fehlern. Stattdessen werden Fehler als reine Werte gesehen, welche das Interface error implementieren und gemäß Konvention über den letzten Platz eines Tupels, welcher der Rückgabetyp ist, zurückgegeben. Des Weiteren werden die Instanzen von komplexen Typen als Referenz zurückgegeben, damit kein unnötiger Kopiervorgang stattfinden muss. Die Implementierung des Interfaces sieht wie folgt aus.

```

1 type BooksService struct {
2     db *sql.DB
3 }
4
5 func (accessor *BooksService) Connect(dataSourceName string, maxOpenConnections
6     uint) error {
7     var openErr error
8     accessor.db, openErr = sql.Open("mysql", dataSourceName)
9
10    if openErr != nil {
11        return openErr
12    }
13
14    _, createErr := accessor.db.Exec(`CREATE TABLE IF NOT EXISTS books(
15        id int auto_increment,
16        isbn text not null,
17        title text not null,
18        author text not null,
19        constraint books_pk primary key (id)
20    )`)
21
22    return createErr
23 }
24
25 func (accessor *BooksService) ReadById(id uint) (*models.Book, error) {
26     result := accessor.db.QueryRow("SELECT * FROM books WHERE id = ?", id)
27
28     var book models.Book

```

```

29   err := result.Scan(&book.Id, &book.Isbn, &book.Title, &book.Author)
30
31   if err != nil {
32     return nil, err
33   } else {
34     return &book, nil
35   }
36 } // Weitere Methoden

```

Listing 55: Go-Code des BooksServices mit der Methode ReadById

Zuerst wird ein neues Struct definiert, welches innerhalb des Moduls erreichbar die Datenbankverbindung hält. Dabei wird dem Struct als Erstes eine Methode zum Verbinden zur Datenbank hinzugefügt. Dieser müssen die Verbindung und die maximale Anzahl der gleichzeitig erlaubten Verbindungen übergeben werden. Außerdem erstellt diese, falls nicht bereits geschehen, die Tabelle für die Bücher. Diese Methode muss folglich immer vor der ersten Verwendung anderer Methoden der Instanz aufgerufen werden. Genauso wie die erste Methode führen alle anderen Methoden direkte SQL-Querys zur Interaktion mit der Datenbank durch. Dabei werden alle Errors sinnvoll behandelt oder, falls dies nicht möglich ist, zurückgegeben.

Der BooksService wird dann in den entsprechenden Controller-Funktionen genutzt. Die Methode GetBook ist nachfolgend abgebildet.

```

1 func GetBook(context *gin.Context) {
2   id, err := strconv.Atoi(context.Param("id")); err != nil || id < 0 {
3     context.AbortWithStatusJSON(http.StatusBadRequest, datatransferobjects.
4       ErrorResponse{Message: "Invalid book id"})
5   } else {
6     book, _ := booksDataAccessor.ReadById(uint(id))
7
8     if book == nil {
9       context.AbortWithStatus(http.StatusNotFound)
10    } else {
11      context.JSON(http.StatusOK, book)
12    }
13  }
14}

```

```

12 }
13 }
14 // Weitere Funktionen

```

Listing 56: Go-Code der Controller-Funktion GetBook

Das meiste des Codes ist recht selbsterklärend. Es ist wichtig zu wissen, dass wenn eine Route von einem Client aufgerufen wird, der entsprechenden Controller-Funktion alle Informationen und Funktionen von Gin in Form eines Context-Objekts übergeben werden. Darüber hinaus muss erwähnt werden, dass jeder Endpunkt-Aufruf eine Controller-Funktion als Goroutine startet. Eine Goroutine ist ein leichtgewichtiger Thread. Goroutinen werden nativ durch die Programmiersprache bereitgestellt und bringen einige Funktionen wie zum Beispiel Channels zur sicheren Kommunikation über verschiedene Goroutinen hinweg mit. Da Gin selbstständig Endpunktaufrufe in Form einer Goroutine startet, muss sich um die Parallelität nicht weiter selbst gekümmert werden.

Die Controller-Funktionen werden dabei in einer eigenen Routerfunktion namens SetupRouter für die entsprechenden Routen gesetzt. Diese wird im Startpunkt des Programmes aufgerufen. Dabei werden die einzelnen Routen auf die Router-Gruppe für den Pfad „/v1/books/“ angewandt (siehe Kapitel 4.4.2).

```

1 func SetupRouter(router *gin.RouterGroup) {
2     router.POST("", controllers.CreateBook)
3     router.GET("", controllers.GetBooks)
4     router.GET("/:id", controllers.GetBook)
5     router.PATCH("/:id", controllers.UpdateBook)
6     router.DELETE("/:id", controllers.DeleteBook)
7 }

```

Listing 57: Go-Code der Funktion SetupRouter

Abruf aller Objekte mit Pagination, Sortierung und Filterung

Zum Abruf mehrerer Bücher mithilfe des BooksServices soll gemäß dem implementierten Interface DataAccessor folgende Methode genutzt werden.

```

1 func (accessor *BooksService) Read(page uint, perPage uint, query string) (uint,
2     *[] models.Book, error) {
3
4     var sqlQuery string = "SELECT * FROM books"
5     var arguments []any = make([]any, 0)
6
7     if query != "" {
8         sqlQuery += " WHERE isbn LIKE ? OR title LIKE ? OR author LIKE ?"
9         arguments = append(arguments, "%" +query+ "%", "%" +query+ "%", "%" +query+ "%")
10    }
11
12    sqlQuery += " ORDER BY books.id ASC LIMIT ?,?"
13    arguments = append(arguments, (page-1)*perPage, perPage)
14
15    rows, err := accessor.db.Query(sqlQuery, arguments...)
16    defer rows.Close()
17    if err != nil {
18        return 0, nil, err
19    }
20
21    var books []models.Book
22    for rows.Next() {
23        var book models.Book
24        rows.Scan(&book.Id, &book.Isbn, &book.Title, &book.Author)
25        books = append(books, book)
26    }
27    if books == nil {
28        books = make([]models.Book, 0)
29    }
30
31    countResult := accessor.db.QueryRow("SELECT COUNT(*) FROM books")
32    var count uint
33    err = countResult.Scan(&count)

```

```

33 if err != nil {
34     return 0, nil, err
35 }
36
37 return count / perPage, &books, nil
38 }
```

Listing 58: Go-Code der Service-Methode Read

Die Methode Read baut einen passenden SQL-Abfragestring auf und führt diesen anschließend aus. Hervorzuheben ist ein für moderne Programmiersprachen eher unübliches Muster: Zum Zuweisen eines aus der Datenbank erhaltenen Werts, wird zuerst eine Variable ohne Wert erzeugt. Die Adresse dieser Variable wird dann an die Scan Methode einer abgeschlossenen Abfrage übergeben, welche den Wert schließlich auf die Variable schreibt.

Die Controller-Funktion für den Endpunkt sieht wie folgt aus und hat keine auffälligeren Teile.

```

1 func GetBooks(context *gin.Context) {
2     var query datatransferobjects.GetAllBooks
3     if result := utilities.BindQuery(context, &query); result {
4         return
5     }
6
7     maximumPage, books, err := booksDataAccessor.Read(uint(query.Page), uint(query.
8         Per_Page), query.Query)
9     if err != nil {
10         context.Error(err)
11         return
12     }
13
14     context.JSON(http.StatusOK, datatransferobjects.BooksResponse{
15         MaximumPage: maximumPage,
16         Items:       books,
17     })
18 }
```

Listing 59: Go-Code der Controller-Funktion GetBooks

Erstellung eines Objekts

Das Erstellen eines Buches ist sowohl im BooksService als auch im Controller recht einfach gehalten. Zunächst soll auf die Methode Create des BooksServices eingegangen werden.

```

1 func (accessor *BooksService) Create(newBook datatransferobjects.CreateBook) (
2     uint, error) {
3
4     result, insertErr := accessor.db.Exec("INSERT INTO books (isbn, title, author)
5         VALUES (?, ?, ?)", newBook.Isbn, newBook.Title, newBook.Author)
6
7     if insertErr != nil {
8
9         return 0, insertErr
10    }
11
12    id, insertedIdErr := result.LastInsertId()
13
14    if insertedIdErr != nil {
15
16        return 0, insertedIdErr
17    } else {
18
19        return uint(id), nil
20    }
21
22 }
```

Listing 60: Go-Code der Service-Methode Create

Dank MariaDB ist es möglich, über die Rückgabe der Datenbankanfrage die zuletzt erstellte Id zu erhalten. Diese kann dann wiederum von der Funktion Create zurückgegeben werden. Abgesehen davon wird hier ein vorhin angesprochener Nachteil der Zerovalues deutlich: Wenn nur auf den ersten Teil, welcher die Id darstellt, des zurückgegebenen Tupels geachtet wird, so ist dieser im Fehlerfall 0, was je nach Datenbank und Index für sich erst einmal ein plausibler Wert sein kann. Erst durch die Berücksichtigung des zweiten Teils des Tupels, dem Error, wird klar, ob 0 tatsächlich die zurückgegebene Id ist oder nicht.

Im Folgenden findet sich noch die Implementierung der entsprechenden Controller-Funktion.

```

1 func CreateBook(context *gin.Context) {
2
3     var createBook datatransferobjects.CreateBook
4
5     if result := utilities.BindJson(context, &createBook); result {
6
7         return
8     }
```

```

5   }
6
7   id, err := booksDataAccessor.Create(createBook)
8
9   if err != nil {
10    context.Error(err)
11    return
12 } else {
13    context.JSON(http.StatusCreated, datatransferobjects.IdResponse{Id: id})
14 }
15 }
```

Listing 61: Go-Code der Controller-Funktion CreateBook

Änderung einzelner Attribute eines Objekts

Das Verändern eines Buches wird mit einer Update-SQL-Anweisung bewerkstelligt. Dabei wird der Query-String, je nachdem, ob ein Attribut gesetzt wurde oder nicht, entsprechend erweitert. Wäre es z. B. auch möglich, einen leeren Titel zu vergeben, käme es hier zu Schwierigkeiten, da nicht zwischen dem ZeroValue und dem eigentlichen Wert unterscheiden werden könnte. Dafür wäre also eine andere Lösung nötig. Nach der Ausführung der Query wird über einen Wahrheitswert zurückgegeben, ob eine Änderung stattfand.

```

1 func (accessor *BooksService) Update(id uint, updateBook datatransferobjects.
2                                     UpdateBook) (bool, error) {
3
4   var sqlQuery string = "UPDATE books SET "
5
6   var sqlUpdateParts []string
7   var arguments []any = make([]any, 0)
8
9   if updateBook.Isbn != "" {
10     sqlUpdateParts = append(sqlUpdateParts, "isbn=?")
11     arguments = append(arguments, updateBook.Isbn)
12   }
13
14   if updateBook.Title != "" {
15     sqlUpdateParts = append(sqlUpdateParts, "title=?")
16     arguments = append(arguments, updateBook.Title)
17 }
```

```

14    }
15
16    if updateBook.Author != "" {
17        sqlUpdateParts = append(sqlUpdateParts, "author=?")
18        arguments = append(arguments, updateBook.Author)
19    }
20
21    if len(arguments) == 0 {
22        return true, nil
23    }
24
25    sqlQuery += strings.Join(sqlUpdateParts, ", ")
26    sqlQuery += " WHERE id=?"
27    arguments = append(arguments, id)
28
29    result, err := accessor.db.Exec(sqlQuery, arguments...)
30
31    if err != nil {
32        return false, err
33    } else {
34        rowsAffected, affectedErr := result.RowsAffected()
35
36        if affectedErr != nil {
37            return false, affectedErr
38        } else {
39            return rowsAffected > 0, nil
40        }
41    }
42}

```

Listing 62: Go-Code der Service-Methode Update

Die Funktion UpdateBook im Controller ist ähnlich wie dessen andere Funktionen programmiert. Zuerst werden eingegebene Werte validiert. Danach werden die entsprechenden Datenbank-Aktionen vorgenommen. Zuletzt wird dem Aufrufenden eine passende Antwort geschickt.

```

1 func UpdateBook(context *gin.Context) {
2     if id, err := strconv.Atoi(context.Param("id")); err != nil || id < 0 {
3         context.AbortWithStatusJSON(http.StatusBadRequest, datatransferobjects.
4             ErrorResponse{Message: "Invalid book id"})
5     } else {
6         var updateBook datatransferobjects.UpdateBook
7         if result := utilities.BindJson(context, &updateBook); result {
8             return
9         }
10
11         _, err = booksDataAccessor.Update(uint(id), updateBook)
12
13         context.JSON(http.StatusNoContent, nil)
14     }
}

```

Listing 63: Go-Code der Controller-Funktion UpdateBook

Löschen eines Objekts

Für das Löschen eines Buches wird die unten abgebildete Methode in dem BooksService bereitgestellt. Ähnlich wie es auch schon beim Aktualisieren eines Buches war, wird zum Wiedergeben, ob eine Änderung in der Datenbank stattfand, ein Wahrheitswert zurückgegeben.

```

1 func (accessor *BooksService) Delete(id uint) (bool, error) {
2     result, err := accessor.db.Exec("DELETE FROM books WHERE id = ?", id)
3
4     if err != nil {
5         return false, err
6     } else {
7         rowsAffected, affectedErr := result.RowsAffected()
8
9         if affectedErr != nil {
10             return false, affectedErr
11         } else {
12             return rowsAffected > 0, nil
}

```

```

13     }
14 }
15 }
```

Listing 64: Go-Code der Service-Methode Delete

Auch die für diese Aufgabe programmierte Funktion DeleteBook im Controller folgt wieder dem Schema Validierung-Datenänderung-Antwort.

```

1 func DeleteBook(context *gin.Context) {
2     if id, err := strconv.Atoi(context.Param("id")); err != nil || id < 0 {
3         context.AbortWithStatusJSON(http.StatusBadRequest, datatransferobjects.
4             ErrorResponse{Message: "Invalid book id"})
5     } else {
6         book, _ := booksDataAccessor.Delete(uint(id))
7         context.JSON(http.StatusNoContent, book)
8     }
}
```

Listing 65: Go-Code der Controller-Funktion DeleteBook

Validierung von erhaltenen Daten

In dieser Umsetzung der Library API wurde auf zwei Weisen validiert. Bei Endpunkten, welche lediglich eine Id in ihrer Route nutzen, wurde diese manuell validiert, da dies insgesamt für weniger Overhead sorgt. Dabei wird sich zuerst der Wert der Id geholt. Danach wird mit strconv.Atoi der Wert von einem Text in eine Zahl umgewandelt und anschließend geprüft, ob alles geklappt hat und der Wert größer als 0 ist. Ansonsten wird die Aktion mit dem Senden eines Fehlers an den Client beendet.

```

1 if id, err := strconv.Atoi(context.Param("id")); err != nil || id < 0 {
2     context.AbortWithStatusJSON(http.StatusBadRequest, datatransferobjects.
3         ErrorResponse{Message: "Invalid book id"})
4 }
```

Listing 66: Go-Code zur Validierung der Id in den Endpunkten

In allen anderen Fällen wird mit Data Transfer Object-Structs und deren Metainformationen sowie selbst geschriebenen Hilfsfunktionen gearbeitet. Im Folgenden wird angenommen, es wird erwartet, dass im Body ein JSON-Objekt des Typs CreateBook geschickt wird. Um den Body zu parsen und zu validieren, muss die ShouldBindJSON-Methode des Context-Objekts aufgerufen und eine Referenz einer Variable des Typs CreateBook übergeben werden. Auf welche Art und Weise welches Attribut validiert wird, wird durch die Metainformationen nach dem Wort „binding“ in der entsprechenden Struct-Definition festgelegt. Es gibt verschiedene ShouldBind-Methoden wie zum Beispiel ShouldBindQuery oder ShouldBindHeader. Im Fehlerfall geben diese einen error zurück, welcher zuerst noch zu dem konkreten Typ ValidationErrors gecastet werden muss. Dieses ValidationErrors-Objekt stellt einige Informationen über die aufgetretenen Fehler bereit. Auf Basis dessen wird das von der Aufgabe verlangte ErrorsResponse-Objekt erstellt. Über das Context-Objekt schickt der Server mit einem BadRequest-Statuscode das ErrorsResponse-Objekt an den Client und gibt false zurück. Gibt die ShouldBind-Methode keinen Fehler zurück, so wurde auf die Variable das geparserte Objekt geschrieben und es wird true zurückgegeben. Da dieser Vorgang unabhängig von der konkreten ShouldBind-Methode ist, aber dennoch verschiedene ShouldBind-Methoden genutzt werden sollen, gibt es eine nicht-exportierte Funktion mit dem Namen bind, welche den beschriebenen Vorgang durchführt und als Übergabeparameter die entsprechen-

de ShouldBind-Methode erhält. Darüber hinaus gibt es dann zwei exportierte Funktionen BindQuery und BindJson, welche lediglich die Funktion bind aufrufen und die passende ShouldBind-Methode übergeben.

```

1 func BindQuery[T any](context *gin.Context, value *T) bool {
2     return bind(context, value, context.ShouldBindQuery)
3 }
4
5 func BindJson[T any](context *gin.Context, value *T) bool {
6     return bind(context, value, context.ShouldBindJSON)
7 }
8
9 func bind[T any](context *gin.Context, value *T, bindFunction func(any) error)
10    bool {
11     if err := bindFunction(value); err != nil {
12         fields, ok := err.(validator.ValidationErrors)
13
14         if ok {
15             var messages []string
16             for _, element := range fields {
17                 messages = append(messages, fmt.Sprintf("%s' failed on the constraint '%s",
18                     strings.ToLower(element.Field()), element.Tag()))
19             }
20
21             context.AbortWithStatusJSON(http.StatusBadRequest,
22                 datatransferobjects.ErrorsResponse{Messages: messages})
23         } else {
24             context.AbortWithStatusJSON(http.StatusInternalServerError,
25                 datatransferobjects.ErrorResponse{Message: "JSON parsing failed"})
26         }
27
28         return true
29     } else {
30         return false
31     }
32 }
```

Listing 67: Go-Code der Methoden BindQuery und BindJson und bind

Da es keinen anderen Mechanismus zum Beenden einer Funktion außer das return-Keyword gibt, geben die Bind-Funktionen einen Wahrheitswert zurück, ob die Validierung erfolgreich war, wobei im Falle von false, die Controller-Funktion mit return beendet wird.

```

1 var createBook datatransferobjects.CreateBook
2 if result := utilities.BindJson(context, &createBook); result {
3     return
4 }
```

Listing 68: Go-Code zur Validierung in einer Controller Funktion

Behandlung von Fehlern

Jede Anfrage eines Clienten besitzt ein eigenes Context-Objekt. In diesem können auch Errors gespeichert werden. Das Speichern eines Errors geschieht mit der Methode Error, wie das unten stehende Listing veranschaulicht.

```

1 func GetBooks(context *gin.Context) {
2     var query datatransferobjects.GetAllBooks
3     if result := utilities.BindQuery(context, &query); result {
4         return
5     }
6
7     maximumPage, books, err := booksDataAccessor.Read(uint(query.Page), uint(query.
8         Per_Page), query.Query)
9     if err != nil {
10         context.Error(err)
11         return
12     }
13
14     context.JSON(http.StatusOK, datatransferobjects.BooksResponse{
15         MaximumPage: maximumPage,
16         Items:       books,
17     })
18 }
```

Listing 69: Go-Code eines Error-Aufrufs

Auf Basis der im Context gespeicherten Errors kann dann eine Middleware geschrieben werden. Gibt es noch eine Middleware, wird durch den Aufruf von Next diese zuerst ausgeführt. Ist dies nicht der Fall, wird ermittelt, ob Fehler im Context gespeichert wurden und, wenn Fehler vorhanden sind, werden diese konateniert und mit dem Statuscode 500 an den Aufrufenden gesendet.

```

1 func ErrorMiddleware(context *gin.Context) {
2     context.Next()
3
4     if len(context.Errors) > 0 {
5         context.JSON(http.StatusInternalServerError, datatransferobjects.
6             ErrorResponse{Message: strings.Join(context.Errors.Errors()[:], " ; ")})
7     }

```

Listing 70: Go-Code der Error Middleware

Das Setzen der Middleware geschieht in der Konfiguration. Dort muss lediglich die Middleware-Funktion mithilfe der Methode Use gesetzt werden. Eine Recovery-Middleware, welche den Server davor bewahrt, bei einem Panic-Error abzustürzen, wird mit dem Erstellen der gin-Engine über die Funktion Default automatisch gesetzt (siehe Listing 71). Bei einem Panic-Error sorgt das Recovery dafür, dass der Server eine Antwort mit dem Statuscode 500 schickt.

Konfiguration des Servers

Das Konfigurieren des Werbservices findet nur an einem zentralen Ort statt, dem Einstiegs-punkt der Anwendung. Im Falle von Go ist das die nachfolgend dargestellte main-Funktion.

```

1 func main() {
2     gin.SetMode(gin.ReleaseMode)
3
4     var engine *gin.Engine = gin.Default()
5     engine.HandleMethodNotAllowed = true
6
7     engine.Use(middlewares.ErrorMiddleware)

```

```

8
9  var booksDataAccessor *services.BooksService = &services.BooksService{}
10 err := booksDataAccessor.Connect("libraryapi:1234@tcp(localhost:21345)/
11   libraryapi?parseTime=true", 150)
12 if err != nil {
13   return
14 }
15 controllers.InjectDependencies(booksDataAccessor)
16
17 var booksGroup *gin.RouterGroup = engine.Group("/v1/books")
18 router.SetupRouter(booksGroup)
19
20 if err := engine.Run(":31483"); err != nil {
21   return
22 }
23 }
```

Listing 71: Go-Code der main-Funktion

Dabei werden auf der einen Seite über Funktionen des gin-Packages Einstellungen getroffen. Auf der anderen Seite muss ein gin-Engine-Objekt erzeugt werden, welches den Webservice darstellt. Über dieses werden weitere Konfigurationen vorgenommen. Ebenso werden diesem die zu verwendenden Middleware übergeben. Des Weiteren wird mithilfe des gin-Engine-Objekts eine Gruppe für Routen, welche mit demselben Pfad beginnen, erzeugt. Unabhängig davon bietet das Objekt noch einige weitere Konfigurationsmöglichkeiten. Zuletzt wird mit ihm auch der Server gestartet.

Unabhängig davon benötigt ein Go-Projekt auch eine mod-Datei, in welcher Informationen über das Go-Modul als solches gesetzt werden. Ein Go-Modul ist dabei eine Sammlung an Go-Packages wie sie auch in der Implementierung genutzt werden. Dabei werden in der mod-Datei unter anderem die benötigte Mindestversion von Go oder die Abhängigkeiten gesammelt niedergeschrieben.

4.4.3 Bewertung

Steven Solleder

Damit wurde der RESTful Webservice mit der letzten Kombination aus Sprache und Framework, Go und Gin, umgesetzt. Es folgt nun dessen Bewertung.

Dokumentation und Wartung

Zur Dokumentation von Gin gibt es drei größere Anlaufpunkte. Zum einen die Website von Gin⁴² selbst, welche neben der eigentlichen Dokumentation noch einen Blog enthält. Als Nächstes das GitHub Repository⁴³. Und zuletzt die Packagebeschreibung auf der offiziellen Module-Platform von Go⁴⁴. Dass es drei offizielle größere Quellen gibt, stiftet Verwirrung. Ebenso ist es nicht hilfreich, dass manche Informationen in gleicher Weise unter allen drei Quellen zu finden sind.

Ein weiterer negativer Aspekt ist, dass die Webseite von Gin scheinbar seit circa einem Jahr nicht mehr gepflegt wird. Denn der letzte Blogeintrag behandelt Gin 1.16, obwohl dieses nun schon in der Version 1.19 vorliegt. Auch sind auf der Webseite einige unvollständige oder sogar leere Seiten zu finden. So steht auf der FAQ-Seite lediglich der Hinweis „TODO: record some frequently asked question from GitHub Issue tab.“. Aus diesem Grund wird sich im Rahmen dieser Bewertung vor allem auf das Repository und die Packagebeschreibung konzentriert.

Das Repository enthält allgemeine Informationen über Gin, welche einen guten Überblick über das Framework verschaffen. Darüber hinaus beinhaltet es einen sehr hilfreichen Quick Start-Guide, welcher für so gut wie alle wesentlichen Aufgaben eines RESTful Webservices Beispiele samt Beschreibungen beinhaltet. Dieser Quick Start Guide ist als äußerst positiv zu bewerten. Schade ist nur, dass dieser Guide etwas versteckt hinter einem Link in dem Repository ist.

⁴²<https://gin-gonic.com/>

⁴³<https://github.com/gin-gonic/gin>

⁴⁴<https://pkg.go.dev/github.com/gin-gonic/gin>

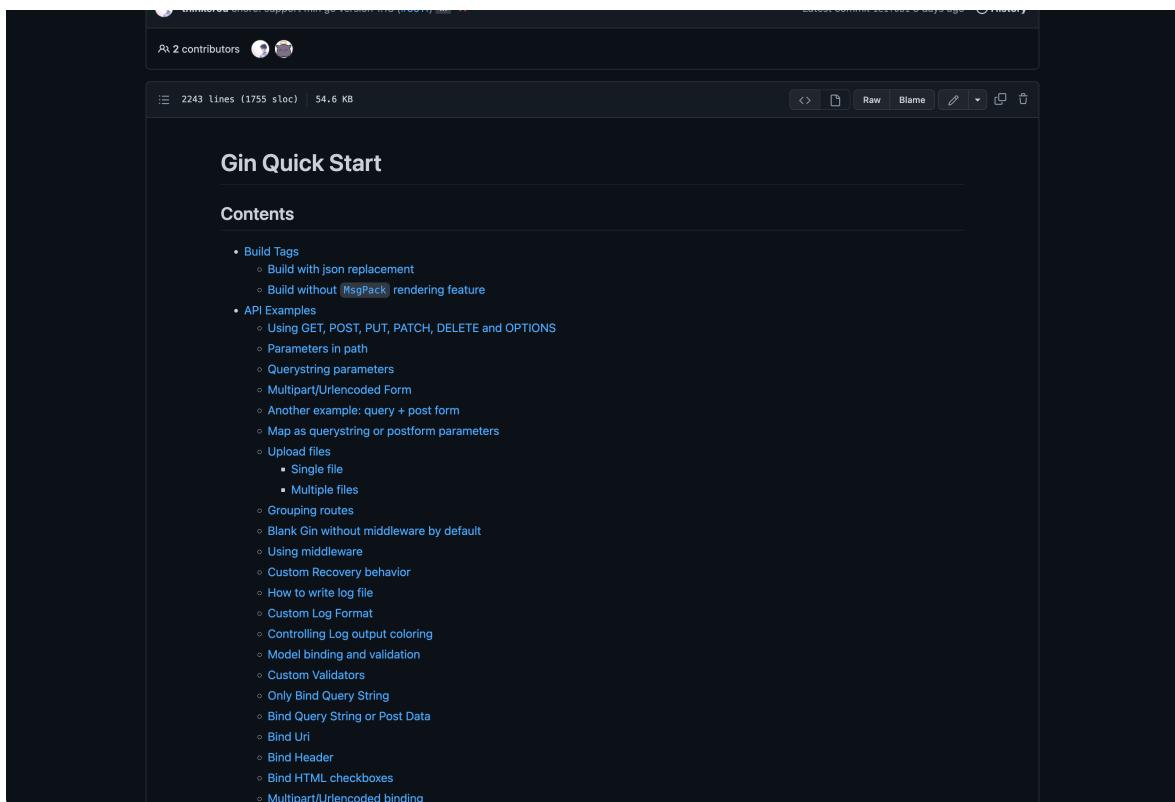


Abbildung 44: Ausschnitt aus dem Gin Quick Start Guide

Hingegen enthält die Packagebeschreibung von Gin zahlreiche weitere Informationen, wenn man als Entwickler tiefer gehendes Verständnis erhalten möchte. Dazu beinhaltet sie eine Auflistung aller von Gin zur Verfügung gestellten Variablen, Konstanten, Funktionen, Typen und deren Methoden. Dabei werden im Falle der Variablen, Konstanten und Typen die Definitionen und im Falle der Funktionen und Methoden deren Signaturen abgebildet. All die gerade genannten Teile von Gin beinhalten stets eine Beschreibung zur näheren Erläuterung. Die Namen all dieser Teile finden sich nach ihrer Art strukturiert auf der linken Seite aufgelistet, sodass sich über diese leicht ein Überblick verschafft und bei Bedarf schnell zu diesen gesprungen werden kann. Am Ende befinden sich auch noch Links zu ausgewählten Quelldateien, wobei diese auch direkt auf GitHub zu finden wären. Zum tieferen Verständnis von Gin oder näheren Erläuterung einzelner Teile ist die Packagebeschreibung somit sehr gut geeignet.

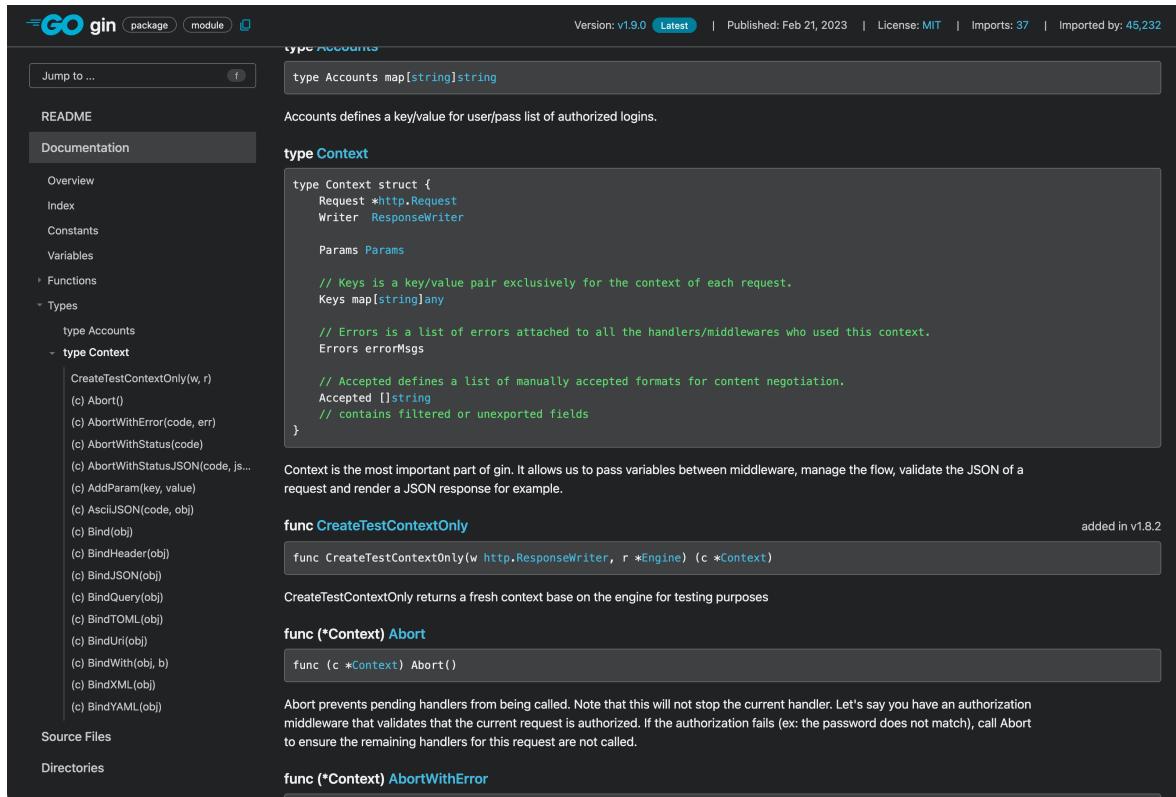


Abbildung 45: Ausschnitt aus der Packagebeschreibung von Gin

Zuletzt soll noch über die Wartung des Frameworks gesprochen werden. Gin erhält jedes Jahr zwei größere Versionen, sprich zum Beispiel Version 1.18 oder 1.19. Dabei versprechen die Macher von Gin jedoch, dass jede größere Version bis zum Erscheinen der übernächsten Major-Version sicherheitskritische Updates bekommt [28]. Das ist sehr positiv zu bewerten, da dadurch sowohl Gin für sich wächst und neue Funktionen erhält, aber ebenso eine größere Zeit lang sich nicht darum gekümmert werden muss, die Version zu wechseln. Dies spart sowohl Kosten als auch Zeit.

Lines of Code

Beim Feststellen der Anzahl der Lines of Codes wurden nur Dateien berücksichtigt, welche die Dateiendung „go“ haben. Bei den anderen Dateien handelt es sich entweder um die Docker Compose Datei oder um andere Konfigurationsdateien. So kommt es folglich unter der Verwendung der cloc-Anwendung zu der folgenden Ausgabe.

Language	files	blank	comment	code
Go	16	78	0	342

Tabelle 6: Ausgabe des Programms cloc bei der Implementierung mit Go und Gin

Notwendigkeit zusätzlicher Bibliotheken

Gin stellt bereits selbst einige Funktionalitäten zur Verfügung, welche zur Umsetzung eines Webservices von Nöten sind, darunter das Routing oder die Validierung. Für Letztere nutzt das Framework Gin eine bereits bestehende, aber fremde Lösung, nämlich das Package go-playground/validator/v10⁴⁵ [26]. Dieses wird aktiv in verschiedene von Gin selbst programmierten Funktionen genutzt.

Bei der Persistierung von Daten bietet Gin hingegen keine eigene Lösung. Zwar gibt es hierfür auch separate ORMs, jedoch sollte im Rahmen dieser Arbeit lediglich das Framework in seiner puren Form untersucht werden. Immerhin kann auf eine Bibliothek von go selbst, nämlich das sql-Package, zurückgegriffen werden, welches alle Funktionen zum Ausführen von Querys und Auslesen der erhaltenen Daten mitliefert. Dieses Package kann jedoch nur genutzt werden, wenn zusätzlich ein Datenbank Connector eingebunden wird, was im Falle dieser Arbeit der mariaDB-Connector ist. So lässt sich zusammenfassend sagen, dass Go erfreulicherweise schon einiges mitbringt, jedoch leider keine optimale Gesamtlösung für das Entwickeln eines RESTful Webservices darstellt.

⁴⁵<https://github.com/go-playground/validator>

Wichtige Architekturprinzipien und besondere Sprachfunktionen

Wie auch schon in den anderen Bewertungskapiteln, soll im Folgenden erst ein Mal ein Überblick über die Architektur mittels folgender Grafik verschafft werden.

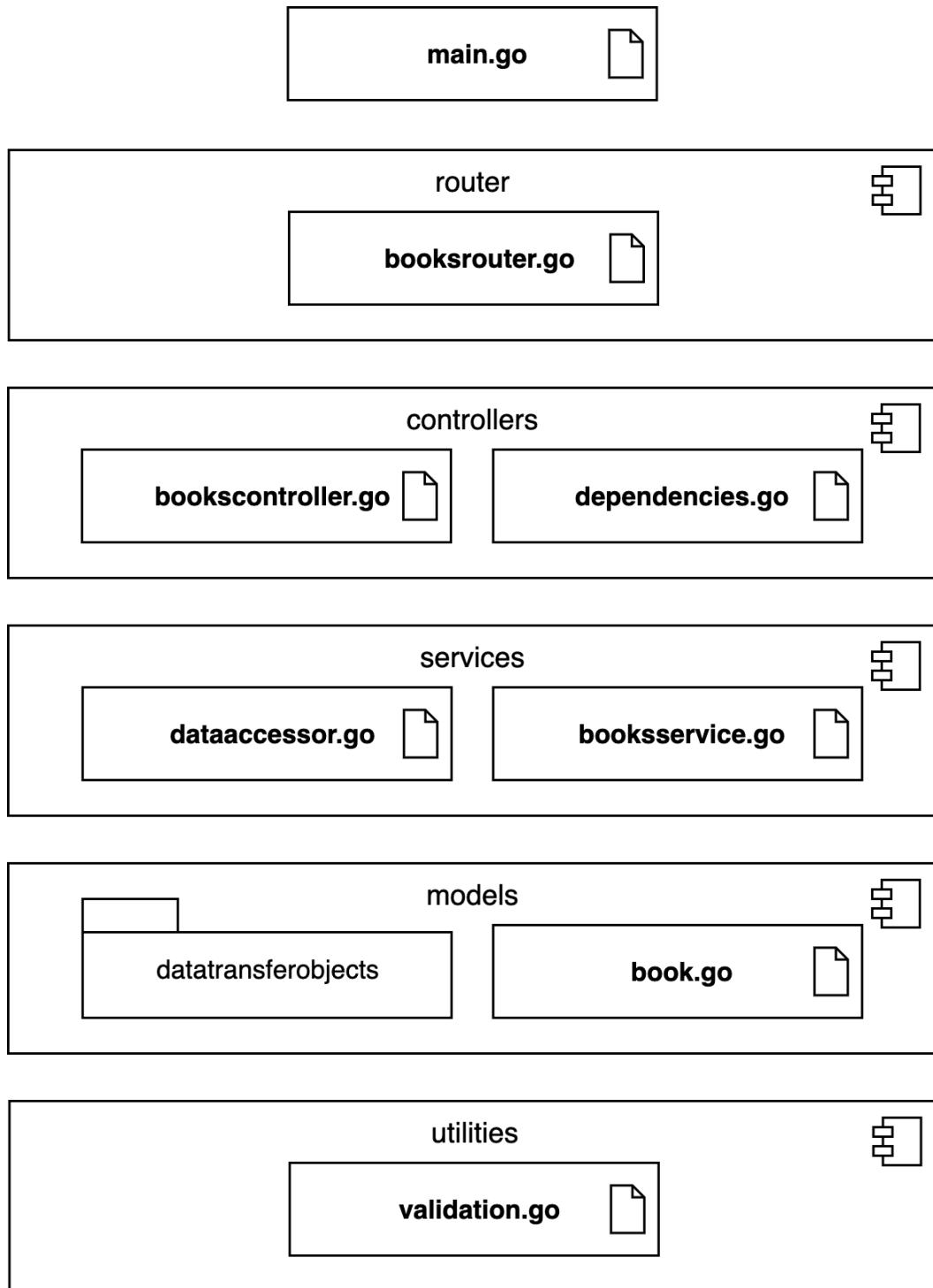


Abbildung 46: Architektur des Webservices mit Gin

Gin setzt in den zur Verfügung stehenden Teilen, wie zum Beispiel seinen Typen oder Funktionen, durchaus zu einem gewissen Maße das Single-Responsibility-Prinzip durch. Jedoch muss der Entwickler immer noch zu einem großen Teil selbst darauf achten, dass auch er gängige Architekturprinzipien einhält, sodass es insbesondere bei größeren Projekten nicht nach einer Weile zu einem größeren Chaos und damit auch zu einem unverständlichen und schwer wartbaren Monolithen kommt. Wo Gin Ansätze für eine gute Architektur liefert und wo der Entwickler sich selbst darum kümmern muss, wird in den folgenden Absätzen näher erläutert.

Beginnend beim Startpunkt, der main-Methode, ist es nötig, ein Gin-Engine-Objekt zu erstellen, um mit Gin zu interagieren. Über dieses quasi allmächtige Objekt laufen dann alle weiteren Operationen entweder mittelbar oder unmittelbar. Insbesondere laufen über das Engine-Objekt das Setzen von Einstellungen, das Zuweisen von Middlewares und mittelbar über eine Group, welche über das Engine-Objekt erstellt wird, das Festlegen von Routen. Dieses allmächtige Objekt widerspricht dem Prinzip der Kapselung leider vollständig. Theoretisch müsste diese Konfiguration nicht komplett in der main-Methode stattfinden, sondern könnte sich auch ungünstigerweise auf verschiedene Stellen verteilen. Das Nutzen einen Composition Roots zur sinnvollen Kapselung der Konfiguration und des Aufbaus der Anwendung muss vom Entwickler selbst ausgehen.

Um dem Engine-Objekt eine Middleware zuzuweisen, muss eine Funktion geschrieben werden, welche von dem Typ HandlerFunc ist. Das für sich ist eine gute Form der Kapselung von Middlewares. HandlerFunc ist eine Funktion mit einem Gin-Context-Objekt als Übergabeparameter und keinem Rückgabetyp. Auch hier besteht wieder ein ähnliches Problem wie bei dem Engine-Objekt, nämlich dass das Context-Objekt über sehr viele Möglichkeiten verfügt, was dem Gedanken der Kapselung widerspricht.

Models werden als Structs definiert, was erstmal für eine gute Kapselung spricht, leider gibt es dabei aber ein paar Probleme. Zum einen können Attribute entweder vollständig oder nur im Package sichtbar und änderbar sein. Das führt dazu, dass sie maximal auf das Package begrenzt, geschützt werden können, wobei es nicht angedacht ist, dass für jedes Struct ein eigenes Packages definiert wird, sodass einige Attribute zwangsläufig nicht so sehr geschützt werden wie sie es eigentlich sollten. Zum anderen gibt es in Go keine

Vererbung, sodass Eigenschaften, wenn nur mittelbar mittels Komposition geteilt werden können.

Wie auf die Datenbank zugegriffen wird, ist aufgrund des fehlenden ORMs vollständig dem Entwickler überlassen. Um hier händisch eine gute Kapselung zu schaffen, wurde ein Interface mit den benötigten Aktionen definiert, welches von einem eigens geschriebenen Typen implementiert wird.

Was die Validierung angeht, müssen dem zu validierenden Attribut eines Structs entsprechende Metainformationen gegeben werden, welche zur Compile Time nicht geprüft werden können, da diese nur als String vorliegen. Das nimmt Sicherheit. Da die Validierung in Form von Strings vorliegt, kann nicht davon gesprochen werden, dass die Validierung für sich gekapselt ist. Hingegen positiv zu bewerten ist, dass mit den entsprechenden Bind-Methoden einkommende Daten mithilfe der gerade erwähnten Structs validiert werden können. Diese Funktionen kapseln also immerhin den Validierungsprozess für sich. Da es verschiedene Bind-Methoden gibt, kann dieser Mechanismus nicht nur zur Validierung des Bodys, sondern zur Validierung jeglicher eintreffender Daten verwendet werden.

Zum Routing kann, wie erwähnt mithilfe des Engine-Objekts ein Gruppen-Objekt erstellt werden, um Routen mit dem gleich beginnenden Pfad zu gruppieren. Dies ist eine positiv zu bewertende Kapselung. Wie dieser Gruppe hingegen die Controller-Funktionen hinzugefügt werden, ist leider wieder vollständig dem Entwickler überlassen. So ist es theoretisch auch möglich, nur anonyme Funktionen zu erstellen, was die Wiederverwendbarkeit unmöglich macht. Da die Controller-Funktionen auch wieder von dem Typ HandlerFunc sind, besteht hier das gleiche Problem wie auch schon bei den Middlewares mit dem zu umfassenden Context-Objekt.

Performance

Zum Schluss soll auf die Performance des aktuell betrachteten Webservices eingegangen werden. Genau wie auch die drei vorherigen Implementierungen der Library API wurde auch die Anwendung mit Go und Gin mit Artillery getestet. Der Report beinhaltete die folgenden beiden Grafiken.

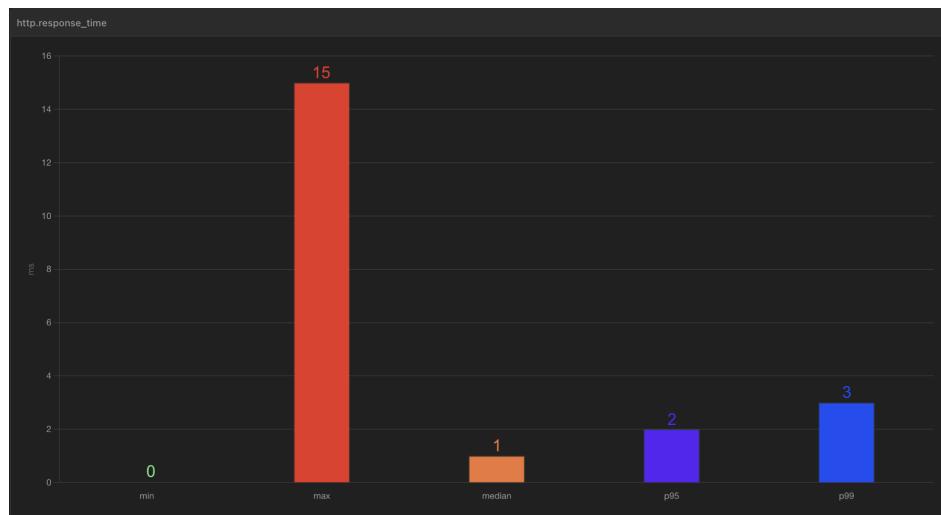


Abbildung 47: Ausgewählte Antwortzeiten für Go mit Gin (Balkenwerte manuell hinzugefügt)

Die oben dargestellte Abbildung stellt verschiedene Antwortzeiten des Services dar. Dabei handelt es sich insbesondere wieder um die minimale und maximale Dauer sowie die Maximalzeiten für 50%, 95% und 99% der Requests. Einige interessante Sessionlängen sind hingegen in der zweiten Abbildung zu sehen. Dabei wird auf dieselben Säulenarten zurückgegriffen.

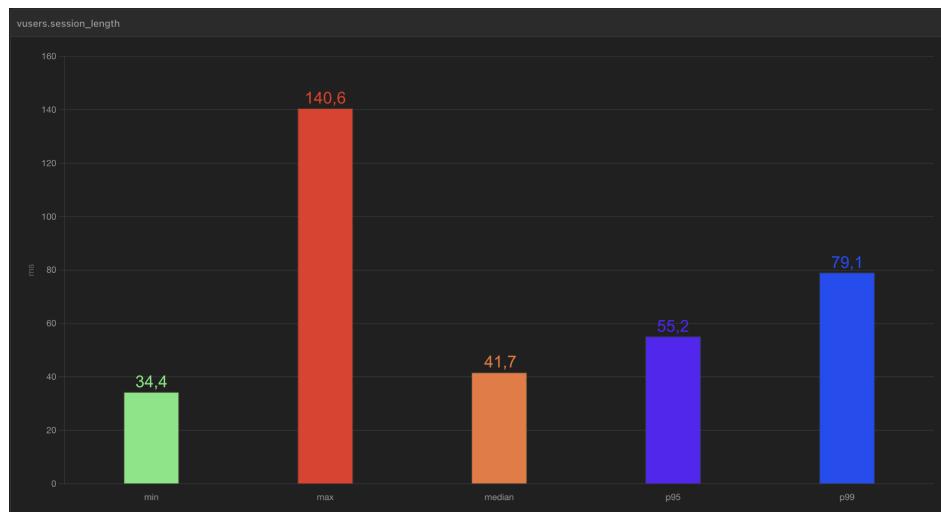


Abbildung 48: Ausgewählte Sessionlängen für Go mit Gin (Balkenwerte manuell hinzugefügt)

5 Auswertung der Erkenntnisse

Steven Solleder

Damit wurde der RESTful Webservice mit jeder der vier ausgewählten Kombinationen aus Programmiersprache und Framework entwickelt sowie bewertet. Nun sollen die gewonnenen Erkenntnisse verglichen und ausgewertet werden, sodass schließlich ein Gesamtfazit getroffen werden kann.

5.1 Gegenüberstellung der Ergebnisse

Steven Solleder, Isabell Waas

Zunächst sollen die Einzelbewertungen der Sprachen und Frameworks miteinander verglichen werden. Dazu stellt die folgende Tabelle in jeder Spalte eine Sprache mit ihrem Framework und in jeder Zeile ein Bewertungskriterium dar. Die verschiedenen Zellen beinhalten dann stets konkrete Werte, beispielsweise die tatsächliche Anzahl der ermittelten Codezeilen im Falle von quantitativ messbaren Kriterien und die wichtigsten Fakten bei qualitativ beurteilbaren Kriterien. Des Weiteren befindet sich auch in jeder Zelle eine Punktzahl, welche sich in dem Bereich 1 bis 10 bewegt.

Zu den Punktzahlen muss gesagt werden, dass es verschiedenste Möglichkeiten gibt, Erkenntnisse in Punkte umzuwandeln und daher keine korrekte beziehungsweise rundum objektive Umwandlung stattfinden kann. Zudem gibt es bei den Kriterien keinen objektiven Maßstab, insbesondere bei den qualitativ zu bewertenden Kriterien, aber auch bei den quantitativ messbaren, da hierfür jegliche existierenden Sprachen und Frameworks herangezogen werden müssten. Stattdessen wird ein relativer Maßstab genutzt, bei dem die drei schlechtesten der ausgewählten Kombinationen aus Sprache und Framework bezogen auf jedes einzelne Kriterium jeweils mit der besten verglichen werden. Die jeweils Beste erhält die Punktzahl 10, die Schlechteste kann minimal die Punktzahl 1 erhalten. Hat die Schlechteste auch gute Eigenschaften, erhält sie eine geringere Punktzahl als die anderen, jedoch mehr als einen Punkt.

In der letzten Zeile der Tabelle ist die jeweils erreichte Gesamtpunktzahl jeder Sprache mit ihrem Framework zu sehen. Diese wird aus der Summe der Punkte, die die Sprache mit ihrem Framework für die einzelnen Bewertungskriterien erhalten hat, berechnet. Hierbei wird

jedes Kriterium als gleich wichtig angesehen, weshalb die Punkte nicht unterschiedlich gewichtet werden.

	C# mit ASP.NET Core	Java mit Spring Boot	TypeScript mit Express.js	Go mit Gin
Dokumentation und Wartung	<u>Dokumentation</u> Umfassend; Vielseitig; Meist Übersichtlich; Etwas verstreut; Manchmal fehlen Beschreibungen; Verlinkt manchmal auf Externes	<u>Dokumentation</u> Umfassend; Vielseitig; Übersichtlich; Teiweise verstreut; Manchmal auf Externes angewiesen	<u>Dokumentation</u> Anfängerfreundlich, Übersichtlich; Wenig tiefgreifend; Verlinkt oft auf Externes	<u>Dokumentation</u> Recht umfangreich; Recht vielseitig; Übersichtlich; Teilweise verstreut; Wenige Teile veraltet
	<u>Wartung</u> Regelmäßig neue Versionen, LTS-Versionen	<u>Wartung</u> Regelmäßig neue Versionen, Normaler und kommerzieller Support	<u>Wartung</u> Regelmäßig Verbesserungen; Selten Hauptversionen	<u>Wartung</u> Regelmäßige Wartung, LTS-Versionen
	10 Punkte	10 Punkte	5 Punkte	7 Punkte
Lines of Code	161 Zeilen	237 Zeilen	449 Zeilen	342 Zeilen
	10 Punkte	7 Punkte	1 Punkte	4 Punkte
Nutzung fremder Bibliotheken	Alles mitgeliefert bis auf Datenbanktreiber	Alles mitgeliefert bis auf Datenbanktreiber	ORM, Datenbanktreiber und Validierung fehlen	ORM und Datenbanktreiber fehlen
	10 Punkte	10 Punkte	6 Punkte	8 Punkte
Architektur und Sprachfeatures	Kapselung wesentlicher Teile stark gefördert; Zahlreiche Sprachfeatures	Kapselung wesentlicher Teile stark gefördert; Mäßig viele Sprachfeatures	Kapselung größtenteils in Entwicklerhand; Gottobjekt; Einige Sprachfeatures	Kapselung meistens in Entwicklerhand; Gottobjekte; Wenige Sprachfeatures
	10 Punkte	8 Punkte	4 Punkte	4 Punkte
Performance	p99 Antwortzeit: 4 ms p99 Sessionlänge: 96,6 ms	p99 Antwortzeit: 5 ms p99 Sessionlänge: 125,2 ms	p99 Antwortzeit: 1.249,1 ms p99 Sessionlänge: 18.963,6 ms	p99 Antwortzeit: 3 ms p99 Sessionlänge: 79,1 ms
	8 Punkte	6 Punkte	1 Punkte	10 Punkte
Summe	48 Punkte	41 Punkte	17 Punkte	33 Punkte

Abbildung 49: Vergleich der Programmiersprachen und Frameworks anhand der fünf Bewertungskriterien

Nun soll auf die einzelnen Kriterien noch kurz eingegangen werden.

Zunächst sollten die Dokumentation und Wartung qualitativ beurteilt werden. Hierzu wurden unter anderem der Umfang und die Benutzerfreundlichkeit der Dokumentationen der Frameworks sowie deren Updatehäufigkeit berücksichtigt. Die Auswertung ergab, dass C# mit ASP.NET Core und Java mit Spring Boot gemeinsam vorne sind. Danach kommen Go mit Gin. TypeScript mit Express.js sind dagegen noch verbesserungswürdig.

Bei den Lines of Code konnten konkrete Werte verwendet werden, was die Bewertung objektiver und damit einfacher machte. Als gut wurde eine geringere Anzahl an Codezeilen angesehen, da dies auf weniger Entwicklungsaufwand hinweist. Hierbei schnitten C# mit ASP.NET Core deutlich am besten ab, gefolgt von Java mit Spring Boot und Go mit Gin.

TypeScript und Express.js lagen auch hier auf dem letzten Platz.

Wie bereits bei dem ersten Kriterium verlief auch die Bewertung der Nutzung fremder Bibliotheken qualitativ. Das Ergebnis fiel dennoch recht klar aus, da C# mit ASP.NET Core und Java mit Spring Boot alle benötigten Funktionalitäten bis auf den bei allen Kandidaten fehlenden Datenbanktreiber selbst mitliefern und so die volle Punktzahl erhalten. Bei Go mit Gin sowie TypeScript mit Express.js sind dagegen ein paar mehr Features nicht vorhanden, sodass diese die hinteren beiden Plätze einnehmen.

Auch die Architektur und Sprachfeatures wurden qualitativ beurteilt. Hierbei wurde vor allem betrachtet, inwieweit Kapselung durch das Framework oder lediglich durch den Entwickler umgesetzt wird. Ebenso war relevant, in welchem Umfang besondere Sprachfunktionen unterstützend eingesetzt werden können. Wie schon zuvor lagen auch hier C# mit ASP.NET Core vorne. Java mit Spring Boot bildeten den zweiten Platz und die anderen beiden Sprachen und Frameworks teilten sich den letzten Platz.

Für die Performance standen in den beiden Diagrammen für die Antwortzeiten beziehungsweise Sessionlängen, die den Artillery-Reports entnommen wurden, mehrere Werte zur Auswahl. Zum Vergleich wurde schließlich der Wert „p99“ herangezogen, welcher die Performance am realistischsten repräsentiert. Grund dafür ist, dass immer bei etwa einem Prozent der Fälle mit auffälligen Randfällen, zum Beispiel einer extrem langen Antwortzeit zu rechnen ist. Für diese kann es zahlreiche Ursachen geben. Jedoch würden diese in der Statistik als „Ausreißer“ bezeichneten Werte das Ergebnis drastisch verfälschen. Daher ist es für das Kriterium Performance vollkommen ausreichend, wenn der für die große Mehrheit der Fälle, welche 99% darstellen, schlechteste Wert für Antwortzeit beziehungsweise Sessionlänge betrachtet wird. Insgesamt waren hier Go mit Gin, insbesondere bei den Sessionlängen, am besten, wobei C# mit ASP.NET Core und Java mit Spring Boot mit jeweils nur kleinen Unterschieden auf dem zweiten und dritten Platz folgen. TypeScript mit Express.js hingegen schnitten auch hier wieder am schlechtesten ab.

Die Gesamtpunktzahlen ergeben damit letztendlich, dass C# mit ASP.NET Core auf Platz 1 liegen, dicht gefolgt von Java mit Spring Boot. Als Drittes kommen Go mit Gin und TypeScript mit Express.js haben mit recht großem Abstand die wenigsten Punkte.

5.2 Gesamtfazit

Steven Solleder

Das Ziel dieser Arbeit war die Ermittlung der für die Umsetzung eines RESTful Webservices geeigneten Programmiersprache sowie des im Rahmen dieser Arbeit besten Frameworks. Wird nun die Tabelle in Abbildung 49 betrachtet, so kann gefolgert werden, dass von den betrachteten Kandidaten die Sprache C# und das Framework ASP.NET Core diesen Platz einnehmen.

Jedoch muss dabei beachtet werden, dass unmöglich eine für jeden Anwendungsfall beste Kombination aus einer Programmiersprache und einem Framework gefunden werden kann. Auch dies veranschaulicht die Tabelle, da sie zeigt, dass beispielsweise für den Bereich Performance Go mit Gin eindeutig vor C# und ASP.NET Core liegen. Daher kann gesagt werden, dass die Kombination C# mit ASP.NET Core nur im Durchschnitt und nur bezogen auf die ausgewählten Bewertungskriterien am besten abschneidet. Allerdings sollte, wenn auf gewisse sachliche Aspekte wie eben zum Beispiel Performance ein besonderes Augenmerk liegt, eine differenziertere Auswahl der Technologien getroffen werden.

Abgesehen davon muss auch gesagt werden, dass in einem Entwicklerteam häufig nicht jede Sprache und jedes Framework beherrscht wird. Auch daher ist es schwierig, eine allgemeine Empfehlung auszusprechen, mit welchen Technologien ein RESTful Webservice programmiert werden soll. Jedoch bietet die Ergebnistabelle in Abbildung 49 einen Anhaltspunkt, um zu entscheiden, ob zum Beispiel lieber Java und Spring Boot oder TypeScript und Express.js genutzt werden soll, wenn die beteiligten Personen beide Sprachen und Frameworks beherrschen.

Literaturverzeichnis

- [1] Philip Ackermann. *Webentwicklung. Das Handbuch für Fullstack-Entwickler*. 1. Auflage. Bonn: Rheinwerk, 2021.
- [2] Christian Braun. *Computernetze kompakt. Eine an der Praxis orientierte Einführung für Studium und Berufspraxis*. 6. Auflage. Berlin, Heidelberg: Springer Vieweg, 2022.
- [3] Pierre Carbonnelle. *PYPL PopularitY of Programming Language*. 2023. URL: <https://pypl.github.io/PYPL.html> (besucht am 14.02.2023).
- [4] MDN contributors. *Content negotiation*. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Content_negotiation#agent-driven_negotiation (besucht am 23.12.2022).
- [5] Shaun Luttin Daniel Roth Rick Anderson. *Overview of ASP.NET Core*. 2023. URL: <https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-7.0> (besucht am 09.03.2023).
- [6] Refsnes Data. *JSON vs XML*. 2023. URL: https://www.w3schools.com/js/js_json_xml.asp (besucht am 16.01.2023).
- [7] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. 2000. URL: <https://roy.gbiv.com/pubs/dissertation/top.htm> (besucht am 23.12.2022).
- [8] Inc. GitHub. *GitHub*. 2023. URL: <https://github.com> (besucht am 14.02.2023).
- [9] Jason Hales, Almantas Karpavicius und Mateus Viegas. *The C# workshop. Kickstart your career as a software developer with C#*. 1. Auflage. Birmingham, UK: Packt Publishing, 2022.
- [10] Artillery Software Inc. *Artillery Docs*. 2023. URL: <https://www.artillery.io/> (besucht am 31.01.2023).
- [11] VMware Inc. *Spring*. 2023. URL: <https://spring.io/> (besucht am 25.02.2023).
- [12] Pivotal Software Inc. *Spring Boot Api*. 2023. URL: <https://docs.spring.io/spring-boot/docs/current/api/index.html> (besucht am 25.02.2023).
- [13] Pivotal Software Inc. *Spring Data Api*. 2023. URL: <https://docs.spring.io/spring-data/jpa/docs/current/api/index.html> (besucht am 18.02.2023).

- [14] Google LLC. *Google Trends Vergleich json und xml*. 2023. URL: <https://trends.google.com/trends/explore?date=2007-01-01%202023-01-01&q=json,xml> (besucht am 16.01.2023).
- [15] Google LLC. *Google Trends Vergleich REST und SOAP*. 2023. URL: <https://trends.google.de/trends/explore?date=2007-01-01%202023-01-01&q=%2Fm%2F03nsxd,%2Fm%2F077dn> (besucht am 19.01.2023).
- [16] Slintel LLC. *Market Share of Github*. 2023. URL: <https://www.slintel.com/tech/source-code-management/github-market-share> (besucht am 14.02.2023).
- [17] Microsoft. *.NET and .NET Core Support Policy*. 2023. URL: <https://dotnet.microsoft.com/en-us/platform/support/policy/dotnet-core> (besucht am 02.03.2023).
- [18] Microsoft. *ASP.NET Core Docs*. 2023. URL: <https://learn.microsoft.com/de-de/aspnet/core/?view=aspnetcore-7.0> (besucht am 15.02.2023).
- [19] Microsoft. *ASP.NET documentation*. 2023. URL: <https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-7.0> (besucht am 09.03.2023).
- [20] Microsoft. *TypeScript Docs*. 2023. URL: <https://www.typescriptlang.org/docs/> (besucht am 14.02.2023).
- [21] Microsoft. *Überblick über C#*. 2022. URL: <https://learn.microsoft.com/de-de/dotnet/csharp/tour-of-csharp/> (besucht am 15.02.2023).
- [22] Microsoft. *Was ist Java Spring Boot?* 2023. URL: <https://azure.microsoft.com/de-de/resources/cloud-computing-dictionary/what-is-java-spring-boot/> (besucht am 13.02.2023).
- [23] Verschiedene GitHub Nutzer. *Express Docs*. 2022. URL: <https://expressjs.com/> (besucht am 14.02.2023).
- [24] Verschiedene GitHub Nutzer. *Gin Docs*. 2022. URL: <https://gin-gonic.com/> (besucht am 14.02.2023).
- [25] Verschiedene GitHub Nutzer. *Gin Web Framework GitHub*. 2023. URL: <https://github.com/gin-gonic/gin> (besucht am 14.02.2023).
- [26] Verschiedene GitHub Nutzer. *Model binding and validation*. 2023. URL: <https://gin-gonic.com/docs/examples/binding-and-validation/> (besucht am 04.03.2023).
- [27] Verschiedene GitHub Nutzer. *npm express*. 2023. URL: <https://expressjs.com/en/guide/routing.html> (besucht am 25.02.2023).

- [28] Verschiedene GitHub Nutzer. *Release History*. 2023. URL: <https://go.dev/doc/devel/release> (besucht am 04.03.2023).
- [29] Verschiedene GitHub Nutzer. *Spring Boot Docs*. 2023. URL: <https://docs.spring.io/spring-boot/docs/current/reference/html/index.html> (besucht am 18.02.2023).
- [30] Verschiedene GitHub Nutzer. *Spring Data Docs*. 2022. URL: <https://docs.spring.io/spring-data/commons/docs/current/reference/html> (besucht am 18.02.2023).
- [31] Verschiedene Wikipedia Nutzer. *C-Sharp*. 2023. URL: <https://de.wikipedia.org/wiki/C-Sharp> (besucht am 15.02.2023).
- [32] Verschiedene Wikipedia Nutzer. *Datei:.NET Core Logo.svg*. 2021. URL: https://de.wikipedia.org/wiki/Datei:.NET_Core_Logo.svg (besucht am 14.02.2023).
- [33] Verschiedene Wikipedia Nutzer. *Datei:Go Logo Blue.svg*. 2019. URL: https://de.wikipedia.org/wiki/Datei:Go_Logo_Blue.svg (besucht am 14.02.2023).
- [34] Verschiedene Wikipedia Nutzer. *Datei:Java-Logo.svg*. 2011. URL: <https://de.wikipedia.org/wiki/Datei:Java-Logo.svg> (besucht am 14.02.2023).
- [35] Verschiedene Wikipedia Nutzer. *Datei:Spring Framework Logo 2018.svg*. 2018. URL: https://de.wikipedia.org/wiki/Datei:Spring_Framework_Logo_2018.svg (besucht am 14.02.2023).
- [36] Verschiedene Wikipedia Nutzer. *Express.js*. 2022. URL: <https://de.wikipedia.org/wiki/Express.js> (besucht am 15.02.2023).
- [37] Verschiedene Wikipedia Nutzer. *File:C Sharp wordmark.svg*. 2019. URL: https://en.wikipedia.org/wiki/File:C_Sharp_wordmark.svg (besucht am 14.02.2023).
- [38] Verschiedene Wikipedia Nutzer. *File:Typescript.svg*. 2022. URL: <https://commons.wikimedia.org/wiki/File:Typescript.svg> (besucht am 14.02.2023).
- [39] Verschiedene Wikipedia Nutzer. *Lines of Code*. 2022. URL: https://de.wikipedia.org/wiki/Lines_of_Code (besucht am 23.08.2023).
- [40] Verschiedene Wikipedia Nutzer. *Representational State Transfer*. 2022. URL: https://de.wikipedia.org/wiki/Representational_State_Transfer (besucht am 15.01.2023).
- [41] Oracle. *Was ist eine relationale Datenbank (RDMBS)?* 2022. URL: <https://www.oracle.com/de/database/what-is-a-relational-database/> (besucht am 19.12.2022).

- [42] Ryan Pinkham. *What Is the Difference Between Swagger and OpenAPI?* 2017. URL: <https://swagger.io/blog/api-strategy/difference-between-swagger-and-openapi/> (besucht am 22.01.2023).
- [43] Inc. Postman. *About Postman.* 2023. URL: <https://www.postman.com/company/about-postman/> (besucht am 09.03.2023).
- [44] IONOS SE. *Golang: Die einfache Programmiersprache aus dem Hause Google.* 2022. URL: <https://www.ionos.de/digitalguide/server/knowhow/golang/> (besucht am 15.02.2023).
- [45] IONOS SE. *SOAP: Das Netzwerkprotokoll erklärt.* 2019. URL: <https://www.ionos.de/digitalguide/websites/web-entwicklung/soap-simple-object-access-protocol/> (besucht am 20.01.2023).
- [46] IONOS SE. *Webservices: Dienste von Maschine zu Maschine.* 2021. URL: <https://www.ionos.de/digitalguide/websites/web-entwicklung/webservice/> (besucht am 19.01.2023).
- [47] Kishori Sharan und Adam L. Davis. *Beginning Java 17 Fundamentals. Object-Oriented Programming in Java 17.* 3. Auflage. Berkeley, CA: Apress, 2022.
- [48] Steven Solleder und Isabell Waas. "Entwicklung eines Bestellsystems und einer Video-kursplattform für die Schwangerschafts-Anwendung Babelli". Hochschule Hof, 2023.
- [49] StatCounter. *Marktanteile der meistgenutzten Suchmaschinen auf dem Desktop nach Page Views weltweit von Januar 2016 bis Januar 2023.* 2023. URL: <https://de.statista.com/statistik/daten/studie/225953/umfrage/die-weltweit-meistgenutzten-suchmaschinen/#:~:text=Im%20weltweiten%20Desktop%2DSuchmaschinenmarkt%20war,rund%2085%20Prozent%20erzielte.> (besucht am 14.02.2023).
- [50] Verschiedene. *curl.1 the man page.* 2023. URL: <https://curl.se/docs/manpage.html> (besucht am 14.02.2023).
- [51] Verschiedene. *Quellcodezeilen.* 2021. URL: https://de.wikibrief.org/wiki/Source_lines_of_code (besucht am 24.01.2023).

Erklärung

Ich erkläre hiermit, dass ich meinen Beitrag zur vorliegenden Gruppenarbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; das gleiche gilt für die von den auf dem Titelblatt der Arbeit genannten Autoren gemeinsam verfassten Teile. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde nach meiner besten Kenntnis bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Hof, den 10. März 2023

Unterschrift

Erklärung

Ich erkläre hiermit, dass ich meinen Beitrag zur vorliegenden Gruppenarbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; das gleiche gilt für die von den auf dem Titelblatt der Arbeit genannten Autoren gemeinsam verfassten Teile. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde nach meiner besten Kenntnis bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Hof, den 10. März 2023

Unterschrift