Bachelorarbeit

Hochschule Merseburg

Fachbereich Ingenieur- und Naturwissenschaften

Matthias Land

Entwicklung eines modular erweiterbaren HTTP Webservers mit standardmäßiger Unterstützung der HTTP Version 1.1

Erstprüfer: Prof. Dr. rer. pol. Uwe Schröter

Zweitprüfer: Prof. Dr.-Ing. Lutz Klimpel

Datum: 17.10.2019

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Das World Wide Web ist eines der größten und meistgenutzten Teile des Internets. Es ist eine riesige Sammlung von Webseiten, welche aus unterschiedlichen Mediendateien bestehen. Die Grundstruktur einer Webseite wird mit Hilfe von HTML maschinenlesbar beschrieben. Zum Ausschmücken einer Webseite können zusätzlich Bilder, Musik oder weitere multimediale Inhalte eingefügt werden.

Damit eine Person eine Webseite abrufen und anzeigen kann, wird ein User Agent benötigt. In dem Fall ist dies ein Webbrowser. Dieser ruft beim Öffnen einer Webseite alle benötigten Dateien ab. Dies geschieht über das HTTP-Protokoll. Der Browser stellt eine HTTP-Anfrage und wartet auf die Antwort des Webservers, der sich um die Verteilung aller Daten kümmert. Ein Webserver nimmt eine Verbindung an, wertet die Anfrage des Nutzers aus und sendet eine Antwort, welche entweder eine Fehlermeldung oder die angeforderte Datei ist.

Es gibt mehrere verschiedene Webserver. Zu den am weitesten verbreiteten gehören Apache, Nginx und Microsoft-IIS [1]. Diese sind allerdings plattformabhängig oder müssen für jedes System neu kompiliert werden und beinhalten oftmals Funktionen, die nicht benötigt werden. Somit können sie die Antwortzeit von Anfragen unnötig verlängern und benötigen zusätzliche Systemressourcen.

Deshalb befasst sich diese Arbeit mit der Entwicklung eines einfachen, plattformunabhängigen Webservers. Dieser soll standardmäßig nur die nötigsten Funktionen bieten, allerdings durch das Einbetten verschiedener Module erweiterbar sein. Dafür ist es nötig, eine API anzubieten, welche das Programmieren von eigenen Modulen ermöglicht.

Damit zwingt man einen Nutzer keine unnötigen Funktionen auf. Zusätzlich kann er weitere benötigte Funktionalitäten einfach hinzufügen.

2 Grundlagen

2.1 Das Hypertext Transfer Protocol

2.1.1 Allgemeines

Ursprünglich wurde das Protokoll 1991 in Form von Version 0.9 eingeführt. [5] Diese Version erlaubte nur das Übertragen von Hypertext-Dokumenten. Im Mai 1996 wurde Version 1.0 als RFC 1945 [6] eingeführt. In dieser Version konnten alle möglichen Dateien zwischen Servern und Clients ausgetauscht werden.
1999 wurde der HTTP 1.1 Standard [2] publiziert, welcher bis 2014 verbessert wurde. Die Hauptverbesserungen von Version 1.1 sind bestehende Verbindungen, das Senden von Dateien an Server und das Fortsetzen von abgebrochenen Verbindungen.
Die aktuelle Version ist 2.0 wurde im Mai 2015 im RFC 7540 [7] eingeführt. Das Hauptziel hiervon war das Beschleunigen und Optimieren der Übertragung. Weiterhin ist es nun möglich, mehrere Abfragen zusammenzufassen, Daten weitergehend zu komprimieren, Inhalte binär kodiert zu übertragen, den Server Datenübertragungen initiieren zu lassen und einiges mehr.

Das Protokoll ist dafür entworfen worden, um Details, wie ein Server funktioniert, zu verbergen. Es bietet lediglich eine bekannte Kommunikationsschnittstelle. Folglich weiß ein Server auch nicht was ein User Agent mit der Antwort anfangen will. [2]
Der Standardport für HTTP ist 80, allerdings werden als Ausweichport oftmals 8080. [33]

Sitzungsdaten

HTTP ist ein zustandsloses Protokoll. Das heißt, jede Anfrage wird als komplett voneinander unabhängig behandelt. Dies sorgt dafür, dass sich ein Server nicht um das Speichern von Sitzungsdaten kümmern muss, was die Geschwindigkeit erhöht und die Komplexität um einiges verringert. [2]
Ursprünglich wurde bei HTTP 1.0 jeweils eine neue Verbindung pro Nachrichtenaustausch verwendet. Ab Version 1.1 können Verbindungen mehrmals benutzt werden, dies sorgt dafür, dass ein Overhead durch das Aufbauen einer TCP-Verbindung vermieden werden kann. Selbst wenn mehrere Abfragen während einer Verbindung übertragen werden wird jede einzelne unabhängig bewertet. [2]

Position im OSI-Modell

HTTP ist ein Protokoll in der Anwendungsschicht, welches direkt auf TCP aufbaut. Somit ist das korrekte und vollständige Ankommen aller Datenpakete garantiert, zumindest solange kein Teilnehmer während der Übertragung ausfällt. [34]

HTTPS

Da beim normalen HTTP alle Daten für gewöhnlich unverschlüsselt übertragen werden könnte jeder, der Zugang zu der Leitung hat, die Daten mitlesen oder verändern. Um dies zu umgehen wurde HTTPS eingeführt. Hierbei werden alle Daten vor dem Senden von beiden Seiten mit Hilfe von TLS verschlüsselt. [2]
Der Standardport für HTTPS ist 443, seltener wird auch 8443 verwendet. [33]

Grundlegende Architektur

Das Protokoll ist Anfrage-Antwort orientiert und basiert auf der Client-Server Architektur. Dabei sind User Agenten, wie zum Beispiel Webbrowser und Crawler die Clients und der Web-Server, wie der Name bereits sagt, der Server. [2]
Der HTTP-Client sendet eine Anfrage an den Server, die eine Anfragemethode, eine URI und die Protokollversion beinhalten. Gefolgt wird dies von einer Reihe von Feldern, die die Anfrage spezifizieren, Information über den Client und einen eventuellen Nachrichtenkörper. Diese Anfrage wird über eine TCP-Verbindung gesendet. [2]
Der Webserver antwortet mit einer Statuszeile, welche die Protokollversion und einen Statuscode beinhaltet. Gefolgt wird diese von Feldern, welche Server- und Antwortsinformation enthält und meist von einem Nachrichtenkörper. [2]

2.1.2 Allgemeiner Syntax

Damit sich mehrere verschiedene Instanzen von Clients und Server untereinander verstehen können, ist es wichtig, dass genau festgelegt wird, wie einzelne Teile einer Nachricht aufgebaut werden müssen. Fehlerhafte Strukturen können dazu führen, dass Nachrichten falsch oder schlimmstenfalls gar nicht verarbeitet werden können.

HTTP-Version

Verschiedene Versionen des gleichen Protokolls werden unterschiedlich gehandhabt. Deshalb ist es wichtig, dass Server, selbst wenn sie eine Version nicht beherrschen, wenigstens wissen, welche Version für diese Nachricht genutzt wird.
Versionen werden durch Haupt- und Nebenversionsnummern gekennzeichnet. Die Version wird durch

HTTP/<Hauptversion>.<Nebenversion>

angegeben. Dabei ist festgelegt, dass Haupt- und Nebenversion durch genau 1 Ziffer angegeben werden, beispielsweise HTTP/1.1, oder HTTP/2.0. [2]

Uniform Resource Identifier

Ein Uniform Resource Identifier, oder abgekürzt URI, ist eine einfache Möglichkeit, die Adresse einer Ressource anzugeben. Bei einer solchen Ressource kann es sich um eine Webseite, einen Webservice, etc. handeln.
URIs werden nicht nur in HTTP gebraucht, sie werden universell genutzt. Man gibt mit ihnen mitunter E-Mail-Adressen und Verlinkungen zum eigenen Filesystem an.
Es ist grundlegend festgelegt, dass eine URI aus einem Schema, einer Authority (Zuständigkeit), einem Pfad, einem Query und einem Fragment besteht. [4]
Diese sind wie folgt zusammengesetzt:

Schema://Authority/Pfad?Query#Fragment

In HTTP werden sie wie folgt benutzt:

http(s)://Host:Port/Pfad?Query

Beispiel: http://maettel.de:8081/index.html?id=25
Falls kein Port angegeben wird, wird Port 80 bei HTTP und Port 443 bei HTTPS angenommen. Mit einem Query können Parameter an einen Server übersendet werden. [2]

Datums- und Zeitangaben

Zeitangaben in HTTP benutzen immer die GMT Zeitzone. Es gibt drei erlaubte Möglichkeiten, die Zeit anzugeben. Davon sind allerdings zwei obsolet und werden nur noch auf Grund von Rückwärtskompatibilität unterstützt. [3]
Das bevorzugte Format ist:

Tag, dd MMM YYYY hh:mm:ss GMT

Dabei ist:	Tag	der englische Tagesname abgekürzt auf 3 Zeichen
	dd	der Tag im Monat
	MMM	der englische Monatsname abgekürzt auf 3 Zeichen
	YYYY	das volle Jahr
	hh	die Stunden im 24-Stunden Format
	mm	die Minuten
	ss	die Sekunden

Beispiele:

Aktuelles Format:	Mon, 07 Jan 2019 20:53:11 GMT
Obsolete Formate:	Monday, 07-Jan-19 20:53:11 GMT
	Mon Jan 7 20:53:11 2019

Cookies

Da in HTTP selbst das Speichern von Sitzungsdaten nicht unterstützt wird, wurden so genannte Cookies mit dem RFC 6265 [10] eingeführt. Cookies sind Daten, die bei jeder Anfrage mit an den Server gesendet werden. Ein Server kann einen Client mitteilen, wenn er Cookies anlegen soll.
Sie bestehen aus mehreren Segmenten, diese sind:

Segmentname	Tatsächliche Darstellung
Cookie-Name + Cookie-Wert	<Cookie-Name>=<Cookie-Feld>
Ablaufdatum	Expires=<Datum>
Maximales Alter	Max-Age=<Zeit in Sekunden>
Domäne	Domäne=<Domänenname>
Pfad	Path=<Pfad>
Sicherheitsfeld	Secure
HTTPOnly-Feld	HttpOnly

Tabelle 1: Cookie-Segmente

Von diesen Feldern ist nur das Namens- und Werte-Feld notwendig, alle weiteren sind optional. Diese Felder werden durch ein Semikolon getrennt.
Der Cookie-Name ist ein eindeutiger Bezeichner eines Cookies. Der Cookie-Wert legt den mit dem Cookie verbundenen Wert als Zeichenfolge, fest.
Das Ablaufdatum legt fest, ab welchen Zeitpunkt ein Cookie gelöscht werden soll. Falls kein Ablaufdatum festgelegt ist, wird der Cookie als Sitzungscookie gekennzeichnet. Er wird also vom Browser gelöscht, sobald er beendet wird. Anstatt eines Ablaufdatums ist es auch möglich, das maximale Alter festzulegen. Dieses legt fest, für wie viele Sekunden der Cookie bestehen bleiben soll. Nach dieser Zeit wird er gelöscht. Wenn man einen Cookie über HTTP entfernen will, muss das Expires-Feld auf einen Zeitpunkt in der Vergangenheit festgelegt oder Max-Age auf 0 gesetzt werden.
Das Domänenfeld gibt an, an welche Domänen der Cookie gesendet werden soll. Wenn dieses Feld nicht gesendet wird, wird das jetzige Host-Feld genutzt.
Der Pfad gibt an, unter welchen Unterseiten der Cookie eingesetzt werden soll. Der jetzige „Ordner“ wird als Pfad angenommen, wenn das Feld fehlt. Das Zeichen „/“ wird als „Ordner“-Separator angesehen.
Das Secure-Feld sorgt dafür, dass dieser Cookie nur über HTTPS verwendet werden soll.
Das HttpOnly-Feld zeigt Browsern, dass dieser Cookie nur durch HTTP-Nachrichten geändert werden kann, nicht durch Skripts, wie beispielsweise Javascript in Browsern.

2.1.3 Aufbau einer HTTP-Nachricht

HTTP-Header

HTTP-Header (englisch für Nachrichtenkopf, folgend nur Header genannt) sind die Kopfzeilen, welche den Beginn jeder HTTP-Nachricht darstellen. Sie beginnen mit einer Startzeile, welche sich bei Anfrage und Antwort unterscheidet. [2]

Nach der Startzeile beginnen die Header-Felder. Diese definieren Parameter, welche an den Empfänger übertragen werden. Header-Felder sind Schlüssel-Wert-Paare und werden durch Doppelpunkte getrennt. Die einzelnen Header-Feldern werden durch einen Zeilenumbruch getrennt. Dieser wird in HTTP durch die ASCII-Zeichen 0D 0A[1] dargestellt. Der Header wird durch eine Leerzeile beendet, also durch zwei aufeinander folgende Zeilenumbrüche. [2]

```
Startzeile
Connection: keep-alive
...
<CRLF>
```

Abbildung 1: Einfaches Beispiel Header

HTTP-Body

Nach dem Header folgt oftmals der Messagebody (englisch für Nachrichtenrumpf, folglich Body genannt). Im Body befinden sich alle Nutzdaten, die an den Empfänger gesendet werden. [2]

Es ist nicht notwendig, dass jede Nachricht einen Body beinhaltet. In manchen Fällen ist es sogar verboten, z.B. bei Code[2] „204 No Content“ (kein Inhalt). [8]

```
Header
<CRLF>
Hello World
```

Abbildung 2: Einfaches Beispiel Body

[1] 0D - Carriage-Return, OA – Line-Feed
[2] Mehr zu Antwortcodes auf Seite 22

Header-Felder

Das HTTP-Protokoll besitzt eine Vielzahl von standardisierten Header-Feldern. In der Praxis werden allerdings auch Felder benutzt, welche nicht standardisiert sind. Diese beginnen für gewöhnlich mit „X-“. Um das Registrieren von neuen standardisierten Feldern kümmert sich IANA [9], die Internet Assigned Numbers Authority.
Es gibt 4 Typen von Header-Feldern [3]: Anfrage-Header, Antwort-Header, Entity-Header und General-Header.

Feld-Art	Beschreibung
Entity-Header	Header, die Information über den Body oder über die angefragte Ressource bereitstellen.
General-Header	Header, die sowohl bei Anfrage und Antwort benutzt werden können.
Anfrage-Header	Header, die den Kontext einer Anfrage konkretisieren oder abhängige Anfragen definieren.
Antwort-Header	Header, die nur in der Antwort genutzt werden können und nichts mit dem Inhalt zu tun haben.

Tabelle 2: Arten der Header-Felder

Header-Felder geben Information zu der Nachricht an, darunter sind Information über den Sender, genutzte Codierungen, Anforderungen an den Empfänger und vieles mehr.

Entity-Header sind Header, die den Inhalt des Nachrichtenkörpers einer Nachricht beschreiben. Sie können sowohl in Anfragen als auch in Antworten benutzt werden. Zu ihnen gehören:

- Content-Length,
- Content-Encoding,
- Content-Language,
- Content-Type. [35]

Das Content-Length-Feld gibt die Länge des Nachrichtenkörpers in Byte an.
Das Encoding-Feld beschreibt, mit welchen Algorithmen der Körper komprimiert wurde. Dies wird getan, um die Länge des Nachrichtenkörpers zu reduzieren und somit die Netzwerklast zu verringern. Es können mehrere Algorithmen angegeben werden, die dann mit einem Komma separiert. Diese müssen in der Reihenfolge, in der sie angewandt wurden, angegeben werden.
Das Language-Feld beschreibt die Sprache der Empfänger, für die die Ressource gedacht ist. Dies heißt allerdings nicht, dass die Ressource selbst in dieser Sprache geschrieben ist.

Das Content-Type-Feld gibt an, welchen Mediatyp die Ressource besitzt. Dieser wird durch einen Mime-Typen angegeben.

Generelle-Header besitzen keinen spezifischen Bezug auf einen Teil der Nachricht und beziehen sich stattdessen auf die Nachricht selbst. Sie geben Information darüber an, wie die Nachricht bearbeitet werden soll.
Auch wenn sie sowohl in Anfragen als auch in Antworten verwendet werden können, werden viele häufig nur in einer der beiden Nachrichten verwendet. Es ist auch möglich, dass sie in Antworten eine andere Bedeutung als in Anfragen haben.
Einige der Generellen-Header sind:

- Cache-Control,
- Connection,
- Date,
- Transfer-Encoding. [35]

Das Cache-Control-Feld [11] gibt Anweisungen an, die von allen Cache-Systemen beachtet werden müssen. Das Feld kann verwendet werden, um Parameter für einen Cache festzulegen oder um spezifische Dokumente von einem Cache anzufordern. Sie werden kommasepariert angegeben. Es handelt sich hierbei um ein Feld, welche andere Parameter bei Anfragen annimmt als bei Antworten. Es gibt aber einige Überschneidungen.
Einige Felder, die bei beiden Nachrichtentypen angewandt werden, sind:

- no-cache es soll kein Cache verwendet werden, ohne die Antwort zu validieren,
- no-store es darf kein Teil der Nachricht gespeichert werden,
- no-transform der Körper der Nachricht darf nicht verändert werden,
- max-age das Alter der Nachricht darf nicht das festgelegte Alter übersteigen. [36]

Das Connection-Feld erlaubt es einem Sender festzulegen, wie nach dem Abschluss der Transaktion mit der Verbindung umgegangen werden soll. Standardmäßig werden Verbindungen in Version 1.0 geschlossen, ab Version 1.1 wurde festgelegt, dass die Verbindung bestehen bleibt.

Eine Verbindung wird beendet, indem ein *Connection: close* an die andere Instanz gesendet wird. Um sie beizubehalten wird *Connection: keep-alive* übersendet. [2]

Transfer-Encoding gibt an, wie der Nachrichtenkörper codiert wurde. Codierung ist nötig, um die Länge des Nachrichtenkörpers zu verringern, oder um dynamisch generierte Ressourcen übertragen zu können. [2]
Transfer-Encoding gibt, anders als Content-Encoding, nicht die Codierung der Ressource an, sondern nur die Codierung der aktuellen Nachricht. Bei einer Verbindung über mehrere Proxys oder Caches können bei jeder Verbindung zwischen Instanzen unterschiedliche Transfer-Encodings genutzt werden. Die Codierung der Ressource bleibt über die gesamte Strecke jedoch gleich. [2]
Für dynamisch generierte Ressourcen wird meist *Transfer-Encoding: chunked* verwendet, da vorher meist nicht bestimmbar ist, wie die Länge des Nachrichtenkörpers ist. Dabei wird der Nachrichtenkörper in mehrere Teile zerlegt. Jeder Chunk besteht aus einer Länge und dessen Inhalt. Die Länge wird hexadezimal angegeben. Abgegrenzt werden diese durch einen Zeilenumbruch. Beendet wird die Nachricht durch eine angegebene Länge von 0 und 2 Zeilenümbrüchen. Zu sehen ist dies in Abbildung 3. [2]

```
6\n\r
Hello \n\r
6\n\r
World!\n\r
0\n\r
\n\r

Decodierte Nachricht: Hello World!
```

Abbildung 3: Beispiel Chunked-Encoding

Chunked ist bei jeder HTTP/1.1 Verbindung zulässig, muss also nicht extra angegeben werden. Es ist ebenfalls möglich, weitere Header-Felder nach dem Ende des letzten Chunks anzugeben. Dies ist nötig, falls diese ebenfalls dynamisch generiert werden. Welche Felder am Schluss übertragen werden, muss allerdings im Header, im Trailer-Feld, angegeben werden. Diese Angabe erfolgt kommasepariert. Ein Client muss in seiner Anfrage allerdings durch das TE[1]-Feld angeben, dass er Trailer akzeptiert. [2]

[1] Das TE-Headerfeld gibt an welche Kodierungen ein Client in einer HTTP-Antwort akzeptiert. [2]

2.1.4 Die HTTP-Anfrage

Ein HTTP-Client sendet eine Anfrage (Request), in Form einer Nachricht, an den Server. Ein Request beinhaltet einen Header und eventuell einen Body. Die Request ist wie folgt aufgebaut:

- Eine Request-Zeile,
- 0 oder mehr Header-Felder (Anfrage/Generelle/Entitäts) gefolgt von einem Zeilenumbruch,
- Leerzeile, um Header von Body abzugrenzen,
- Optionaler Body. [2]

In Abbildung 4 ist eine Request ohne Body zu sehen.

```
GET / HTTP/1.1
Host: maettel.de
Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.119 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: de-DE,de;q=0.9,en-US;q=0.8,en;q=0.7
Cookie: lang=de-DE; darkmode=false
```

Abbildung 4: Beispiel Request Header

Request-Zeile

In der Request-Zeile wird festgelegt, welche Methode die Anfrage nutzt, die URI auf die sich die Anfrage bezieht und welche Version des Protokolls genutzt wird. Die Zeile ist wie folgt aufgebaut:

Methode URI HTTP/<Hauptversion>.<Nebenversion>

Beispiel: GET /index.html HTTP/1.0 [2]

Methoden

Die Methode gibt an, wie der Server die Ressource handhaben soll und welche Antwort der Client von den Server erwartet. Der Methodenname unterscheidet zwischen Groß- und Kleinschreibung und sollte deshalb immer großgeschrieben werden.

In RFC 7231 [8] sind 8 Methoden beschrieben:

Methode	Beschreibung
GET	Jetzige Version der Zielressource transferieren.
HEAD	Dasselbe wie GET es wird aber nur der Header übertragen.
POST	Eine Ressourcenspezifische Verarbeitung an der Zielressource durchführen
PUT	Alle Repräsentationen der Zielressource mit Body ersetzen.
DELETE	Alle Repräsentationen der Zielressource löschen.
CONNECT	Einen Tunnel zum Server, identifiziert mit der Zielressource herstellen.
OPTIONS	Kommunikationsoptionen mit der Ressource beschreiben.
TRACE	Pfad zu einer Ressource testen.

Tabelle 3: In RFC 7231 beschriebene Methoden

Alle Webserver müssen mindestens die Methoden GET und HEAD unterstützen. Alle weiteren Methoden sind optional. Weitere Methoden werden von IANA[1] verwaltet. [8]
Es wird zwischen sicheren und unsicheren Methoden unterschieden. Methoden werden als sicher bezeichnet, wenn sich die Ressource durch das Aufrufen dieser nicht ändert. Zu den sicheren Methoden gehören GET, HEAD, OPTIONS und TRACE. Der Grund für das Unterscheiden von sicheren und unsicheren Methoden ist, dass man automatisiertes Abrufen, zum Beispiel durch Crawler, und Cachen betreiben kann, ohne sich um das Schädigen der Ressourcen sorgen zu müssen. [8]
Weiterhin werden manche Methoden als idempotent bezeichnet. Dies bedeutet, dass mehrmaliges Ausführen derselben identischen Requests genau die gleiche Auswirkung hat wie einmaliges Ausführen. Von den oben genannten Methoden sind PUT, DELETE und alle sicheren Methoden idempotent. Man bezeichnet sie als idempotente Methoden, weil Requests, die auf Grund eines Kommunikationsfehlers die Serverantwort nicht lesen konnten, automatisch wiederholt werden können, ohne Schäden anzurichten. [8]
Pures HTML unterstützt nur das Benutzen von GET- und POST-Methoden, weswegen diese die am häufigsten benutzten Anfragen sind. Die POST-Methode allerdings ebenfalls nur in <form> Elementen [12].

[1] Die Internet Assigned Numbers Authority

Request-URI

Die Request-URI identifiziert die Ressource, auf die sich die Request bezieht. In Tabelle 3 sieht man die verschiedenen Formen, die genutzt werden, um in HTTP eine URI zu spezifizieren. [37]

Form	Beschreibung	Beispiel
*	Der Asterisk beschreibt, dass sich die Anfrage auf den Server direkt bezieht und nicht auf eine spezielle Ressource.	OPTIONS * HTTP/1.0
Absolute URI	Die absolute URI wird bei einer HTTP-Anfrage an einen Proxy benutzt. Der Proxy soll diese Anfrage weiterleiten und die Antwort zurückschicken.	GET http://example.com HTTP/1.1
Absoluter Pfad	Dies ist die meist verwendete Form der Request-URI. Sie wird verwendet, um eine Ressource eines Server zu identifizieren.	GET /galery/img1.png HTTP/1.1 Host: example.com

Tabelle 4: Arten der Request-URI

Request Header-Felder

Ein Client sendet Header-Felder, um mehr Information über den Kontext der Anfrage bereitzustellen, um bedingte Anfragen zu stellen, um bevorzugte Formate der Antwort festzulegen oder um sich zu autorisieren. Request Header-Felder sind in mehrere Kategorien aufgeteilt. [8]

Controls

Controls sind Header-Felder, die das spezifische Handhaben einer Request steuern.
Zu den in RFC 7231 [8] beschriebenen Control-Headern zählen:

Cache-Control, Expect, Host, Max-Forwards, Pragma, Range und TE [8]

Das Feld Expect zeigt an, dass der Client, bevor er eine Anfrage mit (wahrscheinlich großem) Body sendet, sich eine Antwort vom Server wünscht, die angibt, dass er mit seiner Anfrage fortfahren kann. Zurzeit wird nur der Wert „100-continue“ unterstützt, bei allen anderen Werten kann der Server mit einem Fehlercode antworten. [8]
Seit HTTP/1.1 wird das Betreiben mehrerer Hosts auf einem Server unterstützt, das heißt es können mehrere unabhängige Seiten auf einem Server betrieben werden. Um zu unterscheiden, mit welcher Instanz der Client kommunizieren will, wird im Host-Feld angegeben, welche Instanz gewünscht wird. [8]

Bei dem Senden von Requests über Proxies kann man mit dem Max-Forwards-Feld festlegen, wie oft die Anfrage weitergeleitet werden darf. Dies wird allerdings nur von der OPTIONS- und TRACE-Methode unterstützt, bei allen anderen Methoden ist es Servern erlaubt, dieses Feld zu ignorieren. [8]

Conditionals

Durch das Verwenden von bedingten Header-Feldern ist es möglich, eine Reihe von Bedingungen zu setzen, die dafür sorgen, dass die Aktion, die zur Methode gehört, nur ausgeführt wird, wenn die Bedingungen erfüllt sind. Die Bedingungen beziehen sich immer auf vorher erhaltene Validatoren[1] der derzeitigen Ressourcen.
Die festgelegten Felder sind:

- If-Match,
- If-None-Match,
- If-Modified-Since,
- If-Unmodified-Since,
- If-Range. [8]

Content Negotiation

Diese Header-Felder dienen zur Verhandlung der Inhalte der Antwort. Der Client sendet seine bevorzugten Parameter an den Server und führt somit eventuell zu der für den Nutzer am besten passenden Repräsentation der Ressource.
Folgende Felder zur Content Negotiation gibt es:

- Accept,
- Accept-Charset,
- Accept-Encoding,
- Accept-Language. [8]

Die Parameter dieser Felder werden als kommaseparierte Liste angegeben.
Alle Felder der Inhaltsverhandlung unterstützen so genannte Qualitätswerte, abgekürzt durch q. Q-Werte bewerten die relative Wichtung bei der Auswahl der Ressource. Die Ressource mit der höchsten Wichtung wird als am besten passende Ressource ausgewählt und an den Client übergeben. [8]

[1] siehe Seite 27

Q-Werte werden als Fließkommazahl zwischen null und eins mit maximal drei Nachkommastellen angegeben. Wenn kein Q-Wert vorhanden ist, wird ein Wert von eins angenommen. Ein Feld, dessen Parameter mit einem Q-Wert von null bezeichnet ist, zeigt, dass dieser Parameter nicht akzeptiert wird. [8]

Das Accept-Feld wird genutzt, um anzugeben welche Mediatypen, auch MIME-Types genannt, akzeptiert werden. Somit kann festgelegt werden, welche Arten von Dateien der Client annehmen möchte. Diese Arten werden durch Q-Werte, welche vom MIME-Typen durch ein Semikolon getrennt werden, geordnet. [8]

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9
```

Abbildung 5: Accept-Felder mit Q-Werten

Wie in Abbildung 5 zu sehen ist, wird von User Agenten Ressourcen mit dem Typen text/html oder application/xhtml+xml bevorzugt. Falls diese nicht verfügbar sind würde er auch application/xml Dateien annehmen.

Die Accept-Charset-, Accept-Encoding- und Accept-Language-Felder funktionieren auf die gleiche Weise. Zu bemerken ist allerdings, dass fehlende Accept-Felder dem Server zeigen, dass alle Ressourcen akzeptabel sind. [8]

Authentication Credentials

Da manche Ressourcen nicht frei zugänglich sein sollen, wie zum Beispiel Administrationsseiten, ist es nötig, diese nur für User Agenten freizugeben, die dafür authentifiziert sind.
Falls für diese Ressourcen keine manuellen Mechanismen, zum Beispiel über Cookies, implementiert sind, gibt HTTP ab Version 1.1 die Möglichkeit sich zu authentifizieren.
Die dafür entworfenen Felder sind:

- Authorization,
- Proxy-Authorization. [38]

Authorization wird von normalen Ressourcen genutzt und Proxy-Authorization für den Zugriff auf Proxys.
Die Parameter werden im Format <Authentifikationsart> <Anmeldedaten> übersendet.
Die häufigste Authentifikationsart ist Basic. Dabei werden die Daten mit <Nutzername>:<Passwort> angeordnet und mit Base64 codiert. [38]

Request Context

Der Request Context gibt weitere Information über den Absender der Anfrage.
Es wurden 3 Felder definiert:

- From,
- Referer,
- User Agent. [8]

Das From-Feld beinhaltet eine E-Mail-Adresse des Nutzers, der den User Agenten nutzt. Dieses Feld wird sehr selten von Nicht-Crawlern genutzt, da dies unnötig viele Information über den Nutzer preisgibt. [8]
Das Referer-Feld[1] gibt die Adresse der Web-Adresse an, von der aus auf die jetzige Seite zugegriffen wurde. Dies erlaubt es einem Server zu ermitteln, von wo Nutzer auf diesen Server zugreifen und kann diese Daten auswerten. [8]
Der User Agent wird durch das User Agent-Feld angegeben. Darin wird beschrieben, womit die Anfrage gestartet wurde. Eine interessante Beobachtung ist, dass viele Webbrowser nicht genau angeben, um welchen Browser es sich handelt. Stattdessen geben sie eine Listen mehrerer moderner Browser an. [8]
So zeichnet sich Google Chrome z.B. mit „Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.119 Safari/537.36“ aus.

Cookies

In Anfragen werden Cookies, die der User Agent gesetzt hat, gesendet. Diese werden im Cookie-Header-Feld übertragen. Im Cookie-Feld gesendete Cookies beinhalten nur den Namen und den dazugehörigen Wert. Weitere Daten, wie beispielsweise das Ablaufdatum werden nicht übersendet. [10]

Alle Cookies werden, anders als im Set-Cookie-Feld, in einem Header-Feld übertragen. Dabei werden diese durch ein Semikolon und einem Leerzeichen separiert. Der Name und Wert werden durch ein Gleichheitszeichen getrennt. [10]

[1] das Feld wurde im offiziellen Standard falsch geschrieben, eigentlich wäre die richtige Schreibweise Referrer

Range-Requests

Es kann vorkommen, dass eine Datenverbindung während der Übertragung auf Grund von geschlossenen Verbindungen oder Internetverlust abbricht. Wenn ein Client einen Teil einer Resource gespeichert hat, wäre es praktischer, wenn man den restlichen Teil separat anfragen könnte. Ebenso wäre es sinnvoller, nur benötigte Teile einer Ressource anzufragen. Beispielsweise könnte in Videodateien nur der Teil ab eines bestimmten Zeitstempels benötigt werden.

Aus diesem Grund wurden ab HTTP/1.1 Range-Requests, Teil-Antworten und der Mediatyp *multipart/byteranges* eingeführt. Range-Requests sind komplett optional, müssen also nicht von Servern oder Clients unterstützt werden. Es wurde festgelegt, dass bei einer nicht unterstützen Range-Request einfach wie bei einer normalen GET-Anfrage geantwortet werden kann, also mit der gesamten Ressource. [21]

Eine Range-Request ist durch das Range-Header-Feld gekennzeichnet. Dieses Feld besteht aus einer Range-Einheit, für gewöhnlich *bytes*, und einer oder mehrerer Reichweiten. Es ist also möglich, mehrere Teile einer Ressource auf einmal anzufordern. Das Feld startet mit einer Einheit, gefolgt von einem Gleichheitszeichen und danach von kommaseparierten Reichweiten. Reichweiten können auf 3 Arten dargestellt werden. Zu sehen sind diese in Tabelle 5. [22]

Reichweitenart	Beschreibung	Beispiel
`<start>-`	Gibt an, dass die Ressource vom festgelegten Start bis zum tatsächlichen Ende der Ressource übergeben werden soll.	bytes=500-
`<start>-<ende>`	Gibt an, dass die Ressource vom festgelegten Start, bis zum festgelegten Ende übergeben werden soll.	bytes=500-1000, 1500-2000
`-<länge>`	Gibt an, dass die letzten als Länge festgelegten Einheiten übergeben werden sollen.	bytes=-100

Tabelle 5: Reichweitenarten

Das Range-Feld ist nur in GET-Anfragen zulässig. [22]

Es ist möglich, dass sich eine Ressource seit einer vorherigen Anfrage geändert hat. Falls sie sich geändert hat, würde eine Range-Anfrage zu einem unerwünschten Ergebnis führen, etwa einem Download einer fehlerhaften Datei, da mehrere Ressourcen „vermischt“ werden würden.

Um so etwas zu vermeiden, wurde das Header-Feld If-Range eingeführt. Dabei handelt es sich um ein Conditional. Es wird entweder ein Entity-tag, ein Validator mit dem zwischen Repräsentationen einer Ressource unterschieden werden kann, oder ein HTTP-Datum, welches das vorher übergebene Änderungsdatum darstellt, in dem Feld gesendet. Wenn das Feld vorhanden ist, wird vom Server überprüft, ob sich die Ressource geändert hat. Wenn sie sich geändert hat, wird die komplette neue Ressource zurückgegeben, ansonsten wird der angefragte Teil zurückgegeben. [22]

Der Request-Body

Ein Nachrichtenkörper ist nicht nur auf Antworten beschränkt. Es ist auch möglich, eine Anfrage mit einem Körper auszustatten. Was der Körper beinhaltet, ist stark anwendungs- und methodenabhängig. Ein Körper ist allerdings in jeder Methode erlaubt. Benötigt werden Nachrichtenkörper beispielsweise beim Senden von Formulardaten in Webbrowsern oder beim Hochladen von Dateien.
Ein Content-Length- oder Transfer-Encoding-Header-Feld gibt an, dass ein Nachrichtenkörper vorhanden ist. [23]

2.1.5 Die HTTP-Antwort

Der Server nimmt eine Anfrage entgegen, wertet diese aus und sendet dem Client eine passende Antwort (Response). Eine Response ist ähnlich wie eine Request aufgebaut.

```
HTTP/1.1 200 OK
Date: Fri, 22 Mar 2019 18:53:37 GMT
Server: Apache/2.4.18 (Ubuntu)
Vary: Accept-Encoding
Content-Encoding: gzip
Access-Control-Allow-Origin: *
X-Frame-Options: allow-from *
Content-Length: 1048
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
```

Abbildung 6: Beispiel Response Header

Sie besteht aus [13]:

- Statuszeile,
- null oder mehr Header-Feldern,
- Leerzeile zum Abgrenzen von Header und Body,
- optionaler Body.

Wie der Header einer Response aussieht, sieht man in Abbildung 6.

Statuszeile

Die Statuszeile enthält allgemeine Information über das Ergebnis der Anfrage und mit welcher HTTP-Version geantwortet wurde. Das Ergebnis wird durch einen Statuscode und eine Meldung angegeben. Die Zeile ist wie folgt aufgebaut:

HTTP/<Hauptversion>.<Nebenversion> Statuscode Meldung

Die Meldung ist nur eine Beschreibung des Statuscodes und sollte von Clients ignoriert werden. [2]

Statuscodes

Ein Statuscode wird durch 3 Ziffern angegeben und beschreibt das Ergebnis der Anfrage. Sie werden in 5 Klassen aufgeteilt [14]:

- Informative Codes,
- Erfolgscodes,
- Umleitungscodes,
- Client-Fehler,
- Server-Fehler.

Clients müssen nicht alle Codes verstehen, müssen aber zumindest alle Klassen einordnen können. Codes, die nicht bekannt sind, müssen wie der X00-Code behandelt werden, dürfen aber nicht gecached werden. Beispielsweise würde der Code 519[1], solange er nicht bekannt ist, wie der Code 500 behandelt werden.

Informative Codes, welche alle 100-Codes umfassen, werden in Zwischenantworten verwendet. Das heißt als nicht endgültige Antwort, die nur zu Informationszwecken verwendet wird. Zwischenantworten enden mit einer Leerzeile nach dem Header, besitzen also keinen Body.
Die in RFC 7231 [8] festgelegten 100-Codes sind:

- 100 Continue,
- 101 Switching Protocols.

Der Code 100 gibt an, dass der anfängliche Teil der Request erhalten und noch nicht abgelehnt wurde. Die endgültige Antwort folgt am Ende der Verarbeitung der Request.
Der Code 101 zeigt dem Client, dass der Server seine Anfrage versteht und auf das im Upgrade-Feld festgelegte Protokoll wechseln wird. Der Wechsel findet direkt nach einer Leerzeile am Ende des Headers an.

Erfolgscodes, welche die 200-Codes umfassen, zeigen, dass der Server die Anfrage komplett erhalten, ausgeführt und akzeptiert hat.
Einige der in RFC 7231 [8] festgelegten 200-Codes sind:

- 200 OK,
- 201 Created,
- 204 No Content,
- 206 Partial Content.

Der Code 200 zeigt an, dass die Anfrage erfolgreich war. Der Body der Antwort hängt von der Anfragemethode ab.
Der Code 201 gibt an, dass eine oder mehrere Ressourcen erstellt wurden. Der Ort, an der die Ressource erstellt wurde, entspricht der in dem Location-Header-Feld festgelegten URI. Falls kein Location-Feld angegeben ist, wurde sie an der Anfrage-URI erstellt.

[1] Der Code 519 wurde im Standard nicht Festgelegt, er dient hier nur als Beispiel.

Der Code 204 informiert, dass die Anfrage erfolgreich war, aber kein weiterer Inhalt im Body gesendet wurde. Dies wird z.B. bei PUT-Anfragen verwendet, bei der das Senden der Zielressource keinen Sinn ergeben würde.

Der Code 206 sagt aus, dass der Server die Range-Anfrage[1] verstanden hat und gewünschte Teile der Ressource sendet. Die gesendeten Teile müssen im Content-Range-Header-Feld gekennzeichnet werden. Wenn mehr als ein Teil gesendet wird, muss außerdem *Content-Type: multipart/byteranges* angegeben werden.

Umleitungscodes, also alle 300-Codes, geben an, dass der User Agent weitere Aktionen ausführen muss damit die Request erfolgreich ausgeführt werden kann.
Einige der im RFC 7231 [8] festgelegten 300-Codes sind:

- 300 Multiple Choices,
- 301 Moved permanently,
- 302 Found,
- 304 Not Modified,
- 307 Temporary Redirect,
- 308 Permanent Redirect.

Der Code 300 zeigt, dass eine Ressource mehr als eine Repräsentation besitzt. Der Server sendet Information an den User Agenten sodass eine spezielle Repräsentation ausgewählt werden kann. Wenn der Server eine Repräsentation bevorzugt, wird diese im Location-Header-Feld angegeben.
Der Code 301 informiert, dass die Ressource eine neue URI zugewiesen bekam. Alle zukünftigen Anfragen sollen die neue URI referenzieren. Diese URI wird im Location-Feld festgelegt.
Der Code 302 zeigt, dass die Ressource zeitweise eine andere URI benutzt. Die neue URI wird im Location-Feld angegeben.
Der Code 304 sagt aus, dass sich die Ressource seit der letzten Anfrage nicht geändert hat. Dies wird durch vom Client übermittelte Conditionals ermittelt. Der User Agent soll die gespeicherte Repräsentation der Ressource nutzen. Eine 304 Antwort beinhaltet keinen Body. Der Code 307 verhält sich fast genauso wie Code 302, allerdings muss die gleiche Anfragemethode verwendet werden. Code 308 verhält sich genauso zu Code 301.

[1] siehe Seite 20

Die 400-Codes, die Client-Codes, geben an, dass der Client einen Fehler gemacht hat. 400-Antworten sollten eine Ressource übermitteln, die den Fehler beschreibt.
Ein Teil der in RFC 7231 [8] festgelegten 400-Codes sind:

- 400 Bad Request,
- 401 Unauthorized,
- 404 Not Found,
- 405 Method Not Allowed,
- 410 Gone,
- 416 Range Not Satisfiable.

Der Code 400 beschreibt, dass der Server eine Anfrage nicht bearbeiten kann oder wird. Dies kann verschiedene Gründe haben, z.B. Fehler in der Anfragesyntax, fehlerhafte Anfragestruktur, etc.
Der Code 401 sagt aus, dass der Client sich authentifizieren muss, um auf die Ressource zugreifen zu dürfen. Die Authentifizierungsdaten werden im WWW-Authenticate-Header-Feld angegeben.
Der Code 404 ist der bekannteste HTTP-Fehlercode. Er gibt an, dass es keine Ressource an der angefragten URI gibt. Es ist nicht festgelegt, ob dies permanent oder nur temporär ist. Falls die Ressource permanent entfernt wurde, sollte der Code 410 bevorzugt genutzt werden.
Der Code 405 zeigt, dass der Server die Methode zwar kennt, diese aber nicht auf die Ressource angewandt werden darf. Der Server muss eine Liste an erlaubten Methoden über das Header-Feld Allow senden.
Der Code 416 bedeutet, dass eine Range-Anfrage nicht erfüllbar war.

Die Server-Fehler-Codes, die 500-Codes, zeigen, dass ein Fehler im Server auftrat, oder die Anfrage nicht ausgeführt werden kann. Der Server sollte eine Erklärung der Fehlersituation an den Client, im Body der Nachricht, schicken.
Einige der in RFC 7231 [8] festgelegten 500-Codes sind:

- 500 Internal Server Error,
- 501 Not Implemented,
- 503 Service Unavailable,
- 505 HTTP Version Not Supported.

Der Code 500 sagt aus, dass ein unerwarteter Fehler in der Serversoftware auftrat, der dafür sorgte, dass die Anfrage nicht erfüllt werden konnte.

Der Code 501 informiert den Client, dass die in der Anfrage gewählte Methode nicht unterstützt wird. Da HTTP-Server nur die Methoden GET und HEAD unterstützen müssen, tritt dies bei vielen Servern auf, wenn ressourcenverändernde Methoden wie PUT- oder DELETE-Anfragen übermittelt werden.
Der Code 503 sagt aus, dass der Server die Anfrage zurzeit nicht bearbeiten kann. Dies kann daran liegen, dass der Server überlastet ist, oder wegen Instandhaltung nicht verfügbar ist.
Da Clients vorher nicht wissen, welche HTTP-Version ein Server unterstützt, ist es möglich, dass eine Anfrage mit einer nicht kompatiblen Version gestellt wurde. Wenn ein Server für HTTP/1.1 entwickelt wurde und eine HTTP/2.0 Anfrage gestellt wird, kann der Server diese auf Grund großer Unterschiede im Standard nicht durchführen. In diesem Fall wird mit dem Code 505 geantwortet.

Response-Header-Felder

Genau wie eine Anfrage besitzt auch die Antwort eine Vielzahl an Header-Feldern. Mit diesen kann der Server weitere Information über die Antwort preisgeben. Diese Felder sind ebenfalls in mehrere Kategorien aufgeteilt.

Kontrolldaten

Diese Header-Felder können Daten übergeben, die den Statuscode spezifizieren, Cacheanweisungen geben oder dem Client sagen, wohin er weitergeleitet wird. Die wichtigsten dieser Feld sind in Tabelle 6 zu sehen. [26]

Feldname	Beschreibung
Age	Ein von Proxies genutztes Feld, das in Sekunden angibt wie lange die Antwort im Cache war
Cache-Control	Direktiven, die angeben wie ein Cache mit der Antwort umgehen soll
Expires	Eine Datumsangabe, ab wann eine Antwort als veraltet zählt.
Date	Eine Angabe, um welche Zeit die Nachricht erstellt wurde.
Location	Eine Uri auf die verwiesen wird. Die genaue Bedeutung ist Code-abhängig. Bei 300-Codes ist dies die Ressource, auf die weitergeleitet wird.
Retry-After	Gibt an nach wie vielen Sekunden die Anfrage wiederholt werden soll.

Tabelle 6: Kontrolldaten-Header-Felder

Validatoren

Validatoren sind Metadaten, die auf die gewählte Ressource bezogen sind. In Antworten auf Anfragen mit sicheren Methoden beziehen sich die Validatoren auf die angefragte Ressource. Bei Methoden, die Ressourcen ändern, beschreiben Validatoren die neue Repräsentation der Ressource. Es gibt zwei Validator-Felder: ETag und Last-Modified. [27]

Das Last-Modified-Feld beinhaltet den Zeitpunkt, an dem die Ressource das letzte Mal geändert wurde.

Das ETag-Feld (kurz für entity-tag) beinhaltet eine Zeichenkette, die eine spezifische Version einer Ressource beschreibt. Wie ein ETag erstellt wird ist nicht spezifiziert, da eine optimale Erstellung von Anwendung zu Anwendung unterschiedlich sein kann. Es gibt starke und schwache ETags. Schwache ETags beginnen mit einem *W/*. [28]

Respose-Kontext

Diese Felder geben Information über die Zielressource an, die potenziell für spätere Anfragen Nutzen haben. Diese drei Felder sind definiert:

- Accept-Ranges,
- Allow,
- Server.

Das Allow-Feld gibt an, welche Methoden von der Zielressource unterstützt werden. Die erlaubten Methoden werden kommasepariert angegeben.

Das Server-Feld beinhaltet Information über die Software, die der Server nutzt, um die Anfrage zu verarbeiten. [29]

Range-Response

Wenn der Server eine Range-Anfrage verstanden und bearbeitet hat, wird mit dem Code 206 geantwortet. Danach werden im Nachrichtenkörper eine oder mehrere Ressourcenteile gesendet. Wenn es nur einen Teil gibt, muss das Content-Range-Feld im Header vorhanden sein. In dem Fall unterscheidet sich die Antwort kaum von einer normalen 200-Antwort. [24]

Bei einer Antwort mit mehreren Ranges wird die Nachricht allerdings mit dem Content-Type-Feld mit dem Parameter *multipart/byteranges* und einem Separator (boundary in HTTP) versehen. Dadurch wird gekennzeichnet, dass die Nachricht in mehrere Teile aufgespaltet ist. Erkannt werden einzelne Teile durch den Separator. Im Header darf nicht das Feld Content-Range vorhanden sein. [24]

```
HTTP/1.1 206 Partial Content
Content-Type: multipart/byteranges; boundary=SEPARATOR

--SEPARATOR
Content-Type: text/html
Content-Range: bytes 100-125/500

...Inhalt...
--SEPARATOR
Content-Type: text/html
Content-Range: bytes 200-225/500

...Inhalt...
--SEPARATOR--
```

Abbildung 7: Beispiel multipart/byteranges

Jeder Teil der Nachricht startet quasi mit einem eigenen Header. Dieser beginnt mit zwei Bindestrichen, gefolgt von einem Separator und einem CRLF. Dann gibt es mindestens das Felder Content-Range, in dem die Reichweite des folgenden Teils steht. Zusätzlich würde noch das Feld Content-Type dazukommen, wenn der Server einen Typen bei einer 200-Antwort senden würde. Nach einer Leerzeile beginnt der tatsächliche Nachrichtenteil, der durch ein CRLF beendet wird. [24]
Dies wird für jeden Teil wiederholt. Beendet wird ein *multipart/byteranges* durch zwei Bindestriche, den Separator und letztlich noch zwei Bindestriche. Ein Beispiel für eine *multipart/byteranges*-Antwort kann man in Abbildung 7 sehen.

Das Content-Range-Feld wird in einer 206-Antwort gesendet, um anzuzeigen, welcher Teil in der Nachricht vorhanden ist. In einer 416-Antwort gibt er Information über die vorhandene Ressource an. Das Feld startet mit einer Einheit, meist *bytes*, einem Leerzeichen, gefolgt mit der Reichweite, dann einem Schrägstrich und endet mit der Gesamtlänge der Ressource. Wenn die Gesamtlänge nicht festgestellt werden kann, wird stattdessen ein Stern (*) genutzt. Bei einer 416-Antwort wird bei der Reichweite ein Stern genutzt. Beispiele sind in Abbildung 8 zu sehen. [24]

```
Content-Range: bytes 50-150/800
Content-Range: bytes 50-150/*
Content-Range: bytes */800
```

Abbildung 8: Beispiele Content-Range

Ein Server kann angeben, dass Range-Requests unterstützt werden, indem bei einer Antwort das Feld Accept-Ranges angegeben wird. Der Parameter zeigt dabei an, welche Einheiten unterstützt werden. Durch die Einheit *none* wird angegeben, dass keine solchen Requests auf die Ressource unterstützt werden. [25]

3 Problemanalyse

3.1 Grundlegendes

Die entstehende Serversoftware soll auf so vielen Systemen wie möglich, ohne weitere Anpassungen, lauffähig sein. Dafür ist es nötig, eine plattformunabhängige Programmiersprache auszuwählen. Dafür eignet sich Java. Java basiert auf dem Prinzip „compile once run anywhere“. Dafür nötig ist lediglich eine JVM, die für viele Betriebssysteme verfügbar ist. Dies gewährleistet, dass der Server auf vielen Prozessortypen und Betriebssystemen ausführbar ist.
Es wird keine besondere Software zur Datenhaltung benötigt. Es ist lediglich Zugriff auf ein Filesystem nötig, um Einstellungs- und Logdateien anlegen und auslesen zu können.
Der Server soll durch verschiedene Module erweiterbar sein, weswegen es nötig ist Module beim Softwarestart einzulesen und auszuwerten. Für die Entwicklung von Modulen ist es wichtig eine Programmierschnittstelle, also eine API anzubieten.

3.2 Hardwareumgebung

Die Software wird hauptsächlich auf einem Server mit dem Betriebssystem *Ubuntu 16.04* entwickelt. Um zu überprüfen, dass die Software tatsächlich plattformunabhängig arbeitet wird sie ebenfalls auf einem *Windows 10*-System und einem Raspberry Pi 3B getestet.
Das Ubuntu-System besitzt einen 4-Kernprozessor mit je 4 GHz Rechenleistung und 32 GB Arbeitsspeicher. Das Windows-System einen 8-Kernprozessor mit je 3,4 GHz und 16 GB Arbeitsspeicher. Es wird aber nicht ansatzweise so viel Leistung benötigt. Sichtbar ist dies bei dem Raspberry PI. Er besitzt nur 4 Kerne mit je 1 GHz und 1 GB Arbeitsspeicher und ist in der Lage, die Serversoftware auszuführen.

3.3 Anforderungen

Der Server soll grundsätzlich in der Lage sein, HTTP-Verbindungen an einem vom Administrator festgelegten Port anzunehmen und zu verarbeiten. Verbindungen sollen parallel verarbeitet werden, damit mehrere Clients gleichzeitig auf Ressourcen zugreifen können.
Um HTTP-Verbindungen verarbeiten zu können ist es nötig, dass der Server Headerdaten auseinandernehmen und die Anfragemethode interpretieren kann.
Da es möglich sein soll, den Server durch Module zu erweitern, muss es eine API geben. Diese soll sowohl Funktionalitäten als auch Datenstrukturen beinhalten, die vom Server und von Modulen nutzbar sind. Um die Belastung von Festplatten zu verringern soll sie ein stabiles Cachesystem zur Verfügung stellen.

Der Server soll Anfragen sehr schnell verarbeiten können, am besten in unter 500 ms (Datenübertragungszeit nicht einbezogen). Dies ist allerdings stark Hardwareabhängig, weshalb sich die Laufzeit von System zu System unterscheiden wird. Außerdem sollte es in der Lage sein mindestens 200 Verbindungen gleichzeitig verwalten zu können, ohne zu Programmabstürzen oder anderen Problemen zu führen.

Um die API auf die Probe zu stellen, soll der Server durch ein zusätzliches Modul in der Lage sein, php-Dateien durch CGI zu handhaben.

3.4 Tests

Um eine hohe Qualität und Stabilität zu gewährleisten ist es wichtig, dass die Software ausreichend getestet wird. Bei einer relativ großen Software, wie einem HTTP-Server können kleine Fehler zu großen Problemen führen, wenn sie nicht früh genug erkannt werden. Weiterhin kann durch Tests herausgefunden werden, wodurch eventuelle Laufzeitprobleme herbeigeführt werden können. Deshalb ist es wichtig, die Testbarkeit der Software zu gewährleisten.
Es gibt mehrere Testarten, mit denen die Software getestet wird. Eine Art der Tests sind Unit-Tests [15]. Dies sind automatisierte Tests, durch die überprüft wird, ob einzelne Komponenten ihre Aufgabe tatsächlich, wie beabsichtigt, ausführen. Durch das Verwenden von Unit-Tests lassen sich Fehler, die sonst zu späterem Programmversagen führen würden, sofort finden. Weiterhin lässt sich die Laufzeit einzelner Methoden durch

die Nutzung von Unit-Tests bestimmen, wodurch man Performanceprobleme möglicherweise einschränken kann.

Die zweite Art der Tests sind Systemtests [16]. Bei diesen wird die Software direkt getestet. Dafür werden Servereinstellungen vorgenommen und Daten bereitgestellt, die der Server ausliefern soll. Nachfolgend wird in einem Webbrowser die URL zum Server aufgerufen und überprüft, ob die Dateien wie erwartet und unter akzeptablen Laufzeiten ausgeliefert werden. Danach wird entschieden, ob eine Anpassung notwendig ist.

4 Umsetzung

4.1 API

4.1.1 Die Nachrichten-Objekte

HTTP ist ein nachrichtenbasiertes Protokoll. Es gibt zwei Arten von Nachrichten, Responses und Requests. Diese wurden in den Klassen *Request* und *Response* umgesetzt. Die Klasse *Request*, die in Listing 1 zu sehen ist, enthält:

- einen statischen Enumerator, der alle Methoden beinhaltet (Zeile 3),
- die Methode, die in der Anfrage gewählt wurde (Zeile 5),
- die URI und HTTP-Version der Anfrage (Zeile 6) und
- ein *RequestHeader*-Objekt (Zeile 7), welches alle Header-Felder der Anfrage beinhaltet.

```
1  public class Request {
2
3      public static enum Method {GET, HEAD, POST, PUT, DELETE, CONNECT, OPTIONS, TRACE};
4
5      private Method method;
6      private String uri, version;
7      private RequestHeader requestHeader;
8
9      ...
10 }
```

Listing 1: Das Request-Objekt

Zusätzlich gibt es für alle klassenweiten Variablen dazugehörige Getter- und Setter-Methoden.

Parallel dazu wird in der Klasse *Response*, die in Listing 2 zu sehen ist:

- die HTTP-Version, standardmäßig 1.1 (Zeile 3),
- der Antwortcode (Zeile 4),
- ein *ResponseHeader*-Objekt (Zeile 5), das ebenfalls alle Antwort-Header-Felder beinhaltet und
- ein *ResponseOutputStream*[1] (Zeile 7), der zum Ausschreiben der Antworten benötigt wird gespeichert.

Zum Erstellen eines *Response*-Objektes wird ein *BufferedOutputStream* und die Information, ob der Nachrichtenkörper ausgeschrieben werden soll, benötigt. Diese werden im Konstruktor in Zeile 9 übergeben. Der *Stream* wird benötigt, um ein

[1] Siehe: 4.1.5 Ausschreiben von Responses

ResponseOutputStream zu erstellen. Da es bei der HEAD-Methode in HTTP keinen Nachrichtenkörper geben darf wird der Boolean *printBody* benötigt.

```
1  public class Response implements Closeable {
2
3      private String version = "1.1";
4      private String code;
5      private ResponseHeader header = new ResponseHeader();
6
7      private ResponseOutputStream rOut;
8
9      public Response(BufferedOutputStream out, boolean printBody) {
10         rOut = new ResponseOutputStream(out, this, printBody);
11     }
12
13     public ResponseOutputStream getOutputStream() {
14         return rOut;
15     }
16
17     public void close() throws IOException {...}
18
19     public String getStatusLine() {...}
20
21     ...
22 }
```

Listing 2: Das Response-Objekt

Für die Version, den Code und den Response-Header gibt es ebenfalls Getter- und Setter-Methoden. Allein für den ResponseOutputStream gibt es keinen Setter.
Die Klasse implementiert das Interface *Closeable*, damit man direkt auf der *Response* die close()-Methode anwenden kann. Wenn auf einem *Response*-Objekt *close*() angewandt wird, wird automatisch der darunter liegende *ResponseOutputStream* geschlossen.
Sowohl *Response* als auch *Request* beinhalten ein *Header*-Objekt. Diese *Header*-Objekte sind Sammlungen von allen Werten mit dazugehörigen Gettern und Settern. Da dies eine riesige Anzahl an Feldern pro Klasse beinhalten würde, wurden diese in mehrere Klassen aufgeteilt. Daraus ergaben sich 5 *Header*-Klassen:

- HttpHeader,
- GeneralHeader,
- EntityHeader,
- ResponseHeader und
- RequestHeader.

Diese beinhalten jeweils nur Header-Felder, die zu diesen Feldtyp gehören. Diese Klassen erben voneinander. *RequestHeader* und *ResponseHeader* erben von *EntityHeader*, *EntityHeader* erbt von *GeneralHeader*, welches wiederum von *HttpHeader* erbt. Dies sorgt dafür, dass man auf Methoden aller geerbter Klassen zugreifen kann. Somit kann

man beispielsweise über ein *ResponseHeader* auf die Methode *setDate*() von *GeneralHeader* zugreifen.

4.1.2 Die Handler

Der Server soll modular erweiterbar sein, deshalb ist es wichtig gemeinsame Schnittstellen, die von jedem Modul nutzbar sein sollen, bereitzustellen. Eine dieser Schnittstellen sind Handler. Diese sollen von einer zentralen Stelle abruf- und ausführbar sein.

Alle Handler sollen das Interface *Handler*, welches in Listing 3 zu sehen ist, implementieren. Dieses legt fest, dass Handler die Methoden *getHandlerName*() und *getInstance*() implementieren müssen. Die Methode *getHandlerName*() soll einen eindeutigen String zurückgeben, mit dem der Handler gefunden werden kann. Die Methode *getInstance*() ist dafür gedacht, eine neues Objekt einer Handler implementierenden Klasse zurückzugeben.

```
1 public interface Handler {
2     public abstract String getHandlerName();
3     public abstract Handler getInstance();
4 }
```

Listing 3: Das Handler Interface

Normalerweise würde man eine neue Instanz einfach mit dem Schlüsselwort *new* erstellen. Dies ist allerdings nur möglich, wenn die Klasse beim Kompilieren vorhanden war. Da Module erst beim Ausführen der Serversoftware geladen werden, ist dies nicht möglich. Diese Handler müssten dann durch Reflection erstellt werden, was viel langsamer als das Erstellen durch *new* ist [20]. Besonders auffällig wär dies bei einem sehr häufig verwendeten Handler, wie beispielsweise einem Http-Handler. Da bei der Implementation von *getInstance*() bereits zur Kompilierzeit die implementierende Klasse vorhanden ist, kann ein neues Objekt durch *new* erstellt und zurückgegeben werden. Danach ist es lediglich nötig, einmalig beim Laden des Moduls, über Reflection ein Objekt dieser Klasse zu erstellen. Alle späteren Instanzen können dann über *getInstance*() geholt werden.

Das Interface allein bietet aber noch nicht viel mehr als Grundfunktionalitäten. Weitere Funktionalitäten werden durch implementierende, abstrakte Klassen gegeben. Die API beinhaltet zwei abstrakte Klassen, die Handler implementieren, *HttpHandler* und *FileHandler*.

```
1  public abstract class HttpHandler implements Handler {
2
3      public void handleHttp(BufferedOutputStream out, Request request, Map<String, Object> options) throws
   IOException {
4          Response response = new Response(out, request.getMethod() != Request.Method.HEAD);
5
6          handleOptions(request, response, options);
7
8          switch(request.getMethod()) {
9          case GET:
10             doGET(response, request);
11             break;
12         case HEAD:
13             doHEAD(response, request);
14             break;
15         case ...:
16         }
17
18         response.close();
19     }
20     ...
21 }
```

Listing 4: HttpHandler handleHttp()

Die in Listing 4 zu sehende, abstrakte Klasse *HttpHandler* bietet mehrere Methoden an, die direkt zum Verarbeiten von HTTP-Anfragen entwickelt wurden. Die zentrale Methode ist *handleHttp*() in Zeile 3. Diese nimmt einen *BufferedOutputStream*, eine *Request* und eine *Map* mit Optionen an.

Als erstes erstellt sie ein *Response*-Objekt. Dafür wird der *BufferedOutputStream* und die Methode des *Request*-Objekts genutzt. Danach wird die Methode *handleOptions*() aufgerufen. Diese soll alle Optionen, die übergeben wurden, verarbeiten. Danach wird ab Zeile 8, in einem *switch-case*-Block durchgegangen, um welche Methode es sich handelt. Demnach wird die dazugehörige *do*-Methode aufgerufen. Bei der HTTP-Methode GET also *doGET*(), bei POST *doPOST*(), etc.

Dies sorgt für übersichtliche Erweiterbarkeit bei erbenden Klassen. Wenn eine erbende Klasse sich nur um die GET-HTTP-Methode kümmern soll, muss sie nur *doGET*() erweitern. Es bietet aber auch die Möglichkeit, die gesamte Handhabung von *handleHttp*() zu überschreiben, wodurch eine komplett eigene Verhaltensweise entwickelt werden kann.

Nachdem die passende Methode ausgeführt wurde, wird in Zeile 18 die *close*()-Methode der *Response* aufgerufen und diese dadurch beendet.

Wie in Listing 5 zu sehen ist, werfen die *do()*-Methoden Standardmäßig eine *NotImplementedException*, wie beispielsweise in Zeile 4. Die einzige Ausnahme ist *doHEAD*(). In dieser wird in Zeile 8 *doGET*() aufgerufen. Das liegt daran, dass sich die HEAD-Methode in HTTP genauso verhält wie die GET-Methode. Der einzige

Unterschied ist lediglich das Ausschreiben des Nachrichtenkörpers. Darum wurde sich aber schon beim Erstellen des *ResponseOutputStreams* in Listing 4, Zeile 4, gekümmert.

```
1  public abstract class HttpHandler implements Handler {
2
3  public void doGET(Response response, Request request) throws IOException {
4          throw new NotImplementedException();
5      }
6
7      public void doHEAD(Response response, Request request) throws IOException {
8          doGET(response, request);
9      }
10
11     public void doPOST(Response response, Request request) throws IOException {...}
12     public void doPUT(Response response, Request request) throws IOException {...}
13     public void doDELETE(Response response, Request request) throws IOException {...}
14     public void doTRACE(Response response, Request request) throws IOException {...}
15     public void doCONNECT(Response response, Request request) throws IOException {...}
16     public void doOPTIONS(Response response, Request request) throws IOException {...}
17
18     protected void handleOptions(Request request, Response response,  Map<String, Object> options) {}
19   }
```

Listing 5: HttpHandler do()-Methoden

Die *handleOptions*()-Methode ist standardmäßig leer, sie tut also nichts. Sie ist dafür gedacht, dass erbende Klassen weitere Optionen beim Erstellen von Antworten festlegen können, ohne die komplette *handleHttp*()-Methode zu überschreiben.

In Listing 6 ist die abstrakte Klasse *FileHandler* zu sehen. Diese Klasse ist dafür gedacht, sich um das Handhaben von Dateien und Ordnern zu kümmern. Sie bietet die Methoden:

- *handleFile*() zum Behandeln von Dateien (Zeile 12),
- *handleDirectory*() zum Behandeln von Ordnern (Zeile 14),
- *handleRange*() zum Behandeln von Range-Requests (Zeile 17) und
- *handleNoFile()* falls es keine Datei gibt (Zeile 21).

Die Methoden geben alle einen *FinishState* zurück. Dieser gibt an, ob die Behandlung der Abfrage abgeschlossen ist. Das ist nützlich, falls das Bearbeiten einer Anfrage von mehreren Handlern durchgeführt werden soll.

```java
public abstract class FileHandler implements Handler {

    public enum FinishState {FINISHED, CONTINUE}

    protected abstract File getFile(String host, String uri);
    protected File getFile(Request request) {
        return getFile(request.getRequestHeader().getHost(), request.getUri());
    }

    public FinishState handle(Request request, Response response) throws IOException {...}

    protected abstract FinishState handleFile(Request request, Response response, File file)
      throws IOException;
    protected abstract FinishState handleDirectory(Request request, Response response, File file)
      throws IOException;

    protected FinishState handleRange(Request request, Response response, File file) throws IOException {
        return handleFile(request, response, file);
    }

    protected FinishState handleNoFile(Request request, Response response, File file) {
        response.setCode("404");
        response.getHeaders().setHeader("Content-Length", 0);
        return FinishState.FINISHED;
    }
}
```

Listing 6: FileHandler

Da HTTP-Versionen vor 1.1 keine Range-Requests kannten, wird standardmäßig die Methode *handleFile*() in *handleRange*() aufgerufen.

Zusätzlich gibt es zwei Instanzen der Methode *getFile*(),

- einmal in der ein *Request*-Objekt übergeben wird und
- einmal in der 2 Strings übergeben werden, ein Host und die URI.

Die Methoden sind dafür gedacht, die Datei, die angefragt wurde zu bestimmen.

Der Haupteingangspunkt für das Handhaben von Dateien ist die *handle*()-Methode, auf die in Listing 7 weiter eingegangen wird.

```
1  public FinishState handle(Request request, Response response) throws IOException {
2          File file = getFile(request);
3
4          response.getHeaders().setHeader("Date", new HttpDate());
5
6          if (file.exists()) {
7              if (file.isFile()) {
8                  if(request.getRequestHeader().getRanges().isEmpty()) {
9                      return handleFile(request, response, file);
10                 } else {
11                     return handleRange(request, response, file);
12                 }
13             } else if (file.isDirectory()) {
14                 return handleDirectory(request, response, file);
15             } else {
16                 throw new IOException();
17             }
18         } else {
19             return handleNoFile(request, response, file);
20         }
21     }
```

Listing 7: FileHandler handle()

Die *handle*()-Methode nimmt 2 Argumente entgegen, ein *Request*-Objekt und ein *Response*-Objekt. Als erstes wird die Methode *getFile*() in Zeile 2 aufgerufen. Damit wird bestimmt, um welche Datei es sich handelt. Danach wird das Datums-Header-Feld in Zeile 4 durch einen Aufruf der Methoden *getHeaders*().*setDate*() gesetzt.

Folgend wird in Zeile 6 überprüft, ob die Datei existiert und demnach entschieden, was getan werden soll. Wenn die Datei nicht existiert, wird *handleNoFile*() aufgerufen, die eine Antwort mit dem Fehlercode 404 (Not Found) sendet. Wenn sie existiert, wird weiterhin geschaut, ob es sich um eine Datei oder einen Ordner handelt. Bei einem Ordner wird *handleDirectory*() aufgerufen. Falls es eine Datei ist, wird noch zusätzlich unterschieden, ob der Range-Header in der Anfrage gesetzt war. Demnach wird bestimmt, ob die Methode *handleRange*() oder *handleFile*() aufgerufen wird.
Diese Struktur sorgt dafür, dass erbende Klassen nur die *handle*()-Methoden überschreiben müssen, damit sie sich um das Handhaben der jeweiligen Elementtypen kümmern können.

4.1.3 Module

Da der Server modular erweiterbar sein soll, ist es natürlich notwendig, Module erstellen und verwenden zu können. Dafür wurde die abstrakte Klasse *Module* entworfen. Diese soll zur Laufzeit vom Server durch Reflection gefunden und automatisch geladen werden.

Module sollen aus Handlern und Listenern bestehen. Handler sind, wie bereits beschrieben, Schnittstellen, die über eine zentrale Einheit abgerufen werden können. Listener beinhalten Logik zur Behandlung von Netzwerkkommunikationen. So soll beispielsweise ein HTTP-Listener auf einem bestimmten Port auf TCP-Verbindungen warten und eingehende Verbindungen bearbeiten können. Es gibt allerdings keine Pflicht eine Netzwerkkommunikation oder ähnliches in Listener einzubauen. Zu sehen ist das Interface für Listener in Listing 8.

```
1 public interface Listener {
2     public void listen();
3 }
```

Listing 8: Listener Interface

Das Interface bietet nicht mehr als eine *listen*()-Methode. Diese ist lediglich dafür da, dass der Server die Funktionalität der implementierenden Klasse starten kann.

Ein Modul soll mehrere Handler und mehrere Listener besitzen können.

```
1  public abstract class Module {
2
3      public abstract List<Class<? extends Handler>> getHandler();
4      public abstract List<Class<? extends Listener>> getListeners();
5
6      public abstract String getModuleName();
7
8      public abstract void createConfigFile(File configFile) throws IOException;
9      public abstract void loadConfigFile(File configFile) throws IOException;
10 }
```

Listing 9: Abstrakte Module Klasse

Die Klasse, die in Listing 9 zu sehen ist, bietet mehrere Methoden, die für die Nutzung eines Moduls nötig sind. Dazu gehören die Methoden *getHandler*() und *getListeners*(). diese geben jeweils eine Liste von vorhandenen *Handler*- und *Listener*-Klassen zurück. Zu bemerken ist die verwendete Syntax. Es werden in den Zeilen 3 und 4 als Rückgabewert Listen, die mit Klassen bestückt sind, festgelegt. Das *? extends Handler* oder *? extends Listener* bedeutet, dass es sich um Klassen handeln muss, die Handler oder Listener erweitern. Diese Klassen werden später vom Server per Reflection geladen und weiterverwendet.

Module sollten einen eindeutigen Namen besitzen. Dieser wird für die Identifizierung und für das Erstellen von Konfigurationsdateien verwendet. Der Name wird durch den Aufruf der *getModuleName*()-Methode abgefragt.
Und letztlich sollen Module die Möglichkeit bieten, Konfigurationsdateien zu erstellen und auszulesen. In diesen sollen alle möglichen Nutzereinstellungen niedergeschrieben werden. In welchem Format sich die Dateien befinden sollen, ist nicht festgelegt. Es ist aber ratsam, die Einstellungen in einem menschenlesbaren Format, wie JSON oder XML niederzuschreiben. Den Pfad zu der Datei erstellt der Server. Dieser Pfad wird dann in der *createConfigFile*()-Methode übergeben und in dieser wird die Datei erstellt und befüllt. Falls keine Konfigurationen nötig sind, muss keine Datei erstellt werden.
Die Konfigurationen werden dann wieder in der *loadConfigFile*()-Methode eingelesen.

4.1.4 Transfer-Encoding: chunked, der *ChunkedOutputStream*

Da es oftmals vorher nicht möglich ist zu wissen, wie groß der Body einer HTTP-Response genau ist, ist es nötig den Body in Chunks aufzuteilen.
Umgesetzt wurde dies in der Klasse *ChunkedOutputStream*, wie in Listing 10 zu sehen ist. Diese Klasse ist eine Erweiterung der Java-Klasse *BufferedOutputStream*.

```
1  public class ChunkedOutputStream extends BufferedOutputStream {
2
3      private static final byte[] CRLF = "\r\n".getBytes();
4      private boolean closed;
5      private ResponseHeader resp;
6
7      public ChunkedOutputStream(OutputStream out) {...}
8
9      public ChunkedOutputStream(OutputStream out, int size) {...}
10
11     public ChunkedOutputStream(OutputStream out, ResponseHeader resp) {...}
12
13     public ChunkedOutputStream(OutputStream out, int size, ResponseHeader resp) {
14         super(out, size);
15         this.resp = resp;
16     }
17
18     ...
19 }
```

Listing 10: ChunkedOutputStream Konstruktoren

Sie bietet vier Konstruktoren zur Erstellung eines Objektes an (Zeile 7, 9, 11 und 13). Alle Konstruktoren benötigen einen *OutputStream.* Dies ist der darunterliegende Stream, welcher beispielsweise der *OutputStream* einer TCP-Socket-Verbindung sein könnte. Zusätzlich werden in Zeile 9 und 13 noch Konstruktoren mit einer Größe angeboten.

Diese wird direkt an den Superkonstruktor, also den Konstruktor der erweiternden Klasse, weitergegeben. Sie legt die Größe des Buffers fest.
Dieser Buffer ist der Grund, weshalb ein *BufferedOutputStream* erweitert wurde. Er sorgt dafür, dass alle Daten nicht sofort übertragen werden, sondern erst einmal zwischengespeichert werden. Erst wenn der Zwischenspeicher voll ist oder die *flush()*-Methode aufgerufen wird, werden die Daten übertragen. Dies verringert den Overhead, der durch Netzwerkübertragungen entsteht, deutlich. Zusätzlich wird in Zeile 11 und 13 noch die Möglichkeit gegeben, ein *ResponseHeader*-Objekt zu übergeben. Dieser wird zur Übertragung von Trailern benötigt.
Die Klasse speichert (in Zeile 4) das übergebene *ResponseHeader*-objekt (oder null falls keins übergeben wurde) und (in Zeile 5) ob dieser Stream geschlossen wurde. Außerdem wird (Zeile 3) der Einfachheit halber, die Zeichen für den Zeilenumbruch gespeichert.

```
1  public class ChunkedOutputStream extends BufferedOutputStream {
2
3      ...
4
5      @Override
6      public void write(byte[] b, int offset, int length) throws IOException {
7          writeChunkHeader(length);
8          super.write(CRLF, 0, CRLF.length);
9          super.write(b, offset, length);
10         super.write(CRLF, 0, CRLF.length);
11     }
12
13     @Override
14     public void close() throws IOException {
15         if(closed)
16             return;
17         finish();
18         super.close();
19         closed = true;
20     }
21
22     public void finish() throws IOException {
23         writeChunkHeader(0);
24         super.write(CRLF, 0, CRLF.length);
25         writeTrailer();
26         super.write(CRLF, 0, CRLF.length);
27         super.flush();
28     }
29
30     private void writeChunkHeader(int length) throws IOException {
31         byte[] h = Integer.toHexString(length).getBytes();
32         super.write(h, 0, h.length);
33     }
34     ...
35 }
```

Listing 11: ChunkedOutputStream, aufrufbare Methoden

Da es sich um einen *OutputStream* handelt, werden Daten über die *write()*-Methoden geschrieben. Alle angebotenen *write()*-Methoden werden überschrieben, damit keine

falschen Daten übertragen werden. Um die Übersichtlichkeit nicht zu beeinträchtigen, wird in Listing 11 allerdings nur eine Methode (in Zeile 6) gezeigt.
In dieser wird als erstes (in Zeile 7) der Chunk-Header geschrieben. Dafür wird (in Zeile 31) die Länge, die der Nachrichtenteil hat in eine Hexadezimalzahl umgewandelt. Diese wird dann durch den Aufruf der *write()*-Methode des *BufferedOutputStreams* ausgeschrieben. Nachfolgend wird der Zeilenumbruch ausgeschrieben, dann folgt der tatsächliche Bodyinhalt in Zeile 9 und dieser Chunk wird durch einen weiteren Zeilenumbruch beendet.
Abgeschlossen wird die gesamte Nachricht durch den Aufruf der *finish*-Methode. In dieser wird als erstes (in Zeile 23) ein Chunk-Header der Größe 0 ausgeschrieben. Gefolgt von einem Zeilenumbruch. Danach werden in Zeile 25 eventuelle Trailer-Header-Felder ausgeschrieben. Abschließend wird ein letzter Zeilenumbruch geschrieben und die gesamte Nachricht durch die *flush()*-Methode gesendet, wodurch sicher gegangen wird, dass diese versendet wird. Wichtig ist, dass ein Abschließen der Nachricht nicht das Schließen des Streams bedeutet. Wenn die Verbindung geöffnet bleiben soll, darf der Stream nicht geschlossen werden, da der darunterliegende Stream dafür wiederverwendet werden muss.
Falls die Verbindung geschlossen werden soll, kann allerdings die *close()*-Methode verwendet werden. In dieser wird ebenfalls die *finish*-Methode aufgerufen, um zu gewährleisten, dass der Chunk beendet wurde. Danach wird (in Zeile 18) die close()-Methode des *BufferedOutputStreams* aufgerufen. Dies sorgt dafür, dass der darunterliegende Stream geschlossen wird. Abschließend wird der Stream als geschlossen markiert.

```
1  private void writeTrailer() throws IOException {
2          if(resp == null)
3              return;
4
5          HashMap<String, Object> trailers = resp.getTrailer();
6
7          if(trailers.size() == 0)
8              return;
9
10         for(Entry<String, Object> e : trailers.entrySet()) {
11             if(e.getValue() instanceof String) {
12                 String s = (e.getKey() + ": " + e.getValue());
13                 super.write(s.getBytes(), 0, s.getBytes().length);
14             }else if(e.getValue() instanceof List) {
15                 @SuppressWarnings("unchecked")
16                 List<String> trailersList = (List<String>) e.getValue();
17                 String s = (e.getKey() + ": " + makeString(trailersList));
18                 super.write(s.getBytes(), 0, s.getBytes().length);
19             }
20             super.write(CRLF, 0, CRLF.length);
21         }
22     }
```

Listing 12: ChunkedOutputStream writeTrailer

Es ist möglich, bei in Chunks aufgeteilten Nachrichten weitere Header-Felder am Ende der Nachricht anzufügen. Dafür muss es nötig sein, weitere Felder im *ChunkedOutputStream* zu senden. Umgesetzt wurde dies in der Methode *writeTrailer()* (Listing 12). Diese überprüft als erstes, ob ein *ResponseHeader*-Objekt übergeben wurde. Danach werden die Trailer-Felder vom Header-Objekt geholt. Falls es kein *ResponseHeader*-Objekt, oder es keine Trailer-Felder gibt, tut diese Methode nichts und kehrt sofort zurück.

Wenn es Trailer gibt wird über jeden Eintrag in der *trailers*-Map iteriert (Zeile 10). Der Schlüssel jeden Eintrags stellt den Header-Namen und jeder Wert den Inhalt dar. Es ist möglich, als Inhalt sowohl Strings als auch eine Liste von Strings zu definieren. Deshalb gibt es (Zeile 11 und 14) eine Unterscheidung zwischen diesen. Wenn es sich um einen String handelt, wird ein Feld erstellt, indem der Header-Name und der Inhalt kombiniert wird (Zeile 12). Das Feld wird nachfolgend ausgeschrieben.

Bei einer Liste werden alle Elemente dieser kommasepariert zu einem String zusammengefügt (Zeile 17, *makeString()*-Methode). Dieser wird dann mit dem Header-Namen zusammengeführt und ausgeschrieben (Zeile 18). Unabhängig ob es sich um einen String oder eine Liste handelt, wird am Schluss ein Zeilenumbruch ausgeschrieben. Dies wird nun für jedes Element in der *trailers*-Map wiederholt.

4.1.5 Ausschreiben von Responses

Da das Ausschreiben von Responses ein wichtiger Bestandteil von HTTP ist, ist es nötig, eine Klasse anzubieten, die diese Funktion übernimmt. Diese Funktion wird von der Klasse *ResponseOutputStream* (zu sehen in Listing 13) übernommen.

```
1  public class ResponseOutputStream implements Flushable, Closeable, AutoCloseable {
2
3      private final Response response;
4      private final BufferedOutputStream out;
5      private final boolean printBody;
6
7      public ResponseOutputStream(BufferedOutputStream out, Response r, boolean printBody) {
8          this.out = out;
9          this.response = r;
10         this.printBody = printBody;
11     }
12
13     ...
14 }
```

Listing 13: ResponseOutputStream

Wie in Zeile 1 zu sehen ist, handelt es sich trotz des Namens nicht um eine Klasse, die *OutputStream* erweitert. Der Grund dafür ist, dass sich diese Klasse nicht selbst um das Ausschreiben kümmert. Dafür werden intern andere *OutputStreams* verwendet.
Die Klasse bietet einen Konstruktor (Zeile 7). Dieser nimmt 3 Parameter an:

- einen BufferedOutputStream, der zum Ausschreiben benutzt wird,
- eine Response, die Information über die Antwort enthält und
- einen Boolean, der festlegt ob der Body der Antwort ausgeschrieben werden soll[1].

Diese 3 Parameter werden für die spätere Verwendung gespeichert.
Obwohl der *ResponseOutputStream* kein richtiger *OutputStream* ist, soll er genauso verwendet werden können. Deshalb werden die gewohnten *write()*-Methoden und die *flush()*- und *close()*-Methode der Interfaces *Flushable*, *Closeable* und *AutoClosable* angeboten.

[1] In Anfragen mit der HEAD-Methode wird kein Body ausgeschrieben.

```
1  public class ResponseOutputStream implements Flushable, Closeable, AutoCloseable {
2
3      private ChunkedOutputStream chunkedOut;
4      private boolean headerSent, chunked;
5
6      public void write(byte b) throws IOException {
7          commitHeader();
8
9          if(printBody) {
10             if(chunked) {
11                 chunkedOut.write(b);
12             }else {
13                 out.write(b);
14             }
15         }
16     }
17
18     public void write(String s) throws IOException {
19         write(s.getBytes());
20     }
21
22     public void write(byte[] b) throws IOException {...}
23
24     public void write(byte[] b, int offset, int length) throws IOException {...}
25
26     ...
27 }
```

Listing 14: ResponseOutputStream write

Insgesamt werden vier *write()*-Methoden angeboten (Listing 14, Zeile 6, 18, 22, 24). Diese rufen immer die *write()*-Methode des darunter liegenden Streams auf. Ausgenommen davon ist die *write()*-Methode, die einen String annimmt. Diese wandelt den String in Bytes um (Zeile 19) und ruft die Methode auf, die ein Byte-Array annimmt. Die restlichen drei Methoden versuchen als erstes, die Header durch den Aufruf der *commitHeader()*-Methode zu senden. Danach wird überprüft, ob der Nachrichtenkörper gesendet werden soll. Falls nicht, wird nur der Header gesendet. Falls er gesendet werden soll, wird geschaut, ob die Nachricht in Chunks geteilt werden soll. Soll er aufgeteilt werden, wird der *ChunkedOutputStream* genutzt (Zeile 11), ansonsten wird der im Konstruktor übergebene *BufferedOutputStream* genutzt (Zeile 13).

Die *commitHeader()*-Methode, zu sehen in Listing 15, kümmert sich um das einmalige Senden der Header-Felder. Als erstes wird überprüft, ob der Header bereits gesendet wurde (Zeile 4). Die Methode kehrt sofort zurück, wenn er gesendet wurde. Danach wird in Zeile 7 überprüft, ob der Körper in Chunks aufgeteilt werden kann, indem die Methode *isChunkable()* aufgerufen wird. Wenn es aufgeteilt werden kann, wird *Transfer-Encoding: chunked* zum Header hinzugefügt, die Variable *chunked* auf wahr gesetzt und ein *ChunkedOutputStream* erzeugt. Danach wird in Zeile 13 ein String erzeugt, der den

kompletten Header beinhaltet. Dieser wird dann ausgeschrieben und die Variable *headerSent* auf wahr gesetzt.

```
1  public class ResponseOutputStream implements Flushable, Closeable, AutoCloseable {
2
3      private void commitHeader() throws IOException {
4          if(headerSent)
5              return;
6
7          if(isChunkable()) {
8              response.getHeaders().addTransferEncoding("chunked");
9              chunked = true;
10             chunkedOut = new ChunkedOutputStream(out, response.getHeaders());
11         }
12
13         String headerText = response.getStatusLine() + "\r\n" + response.getHeaders().toString() + "\r\n";
14
15         out.write(headerText.getBytes());
16         out.flush();
17         headerSent = true;
18     }
19
20     private boolean isChunkable() {
21         if(printBody && response.getHeaders().getContentLength() == -1)
22             if(!response.getVersion().equals("1.0") && !response.getVersion().equals("0.9"))
23                 return true;
24
25         return false;
26     }
27
28     ...
29 }
```

Listing 15: ResponseOutputStream commitHeader

Die *isChunkable()*-Methode bestimmt ob die Nachricht, die der *ResponseOutputStream* ausschreibt, in Chunks aufgeteilt werden kann. Zu sehen ist diese in Listing 15. Dafür wird überprüft, ob der *Content-Length*-Header festgelegt wurde und ob der Nachrichtenkörper ausgeschrieben werden soll. Zusätzlich schaut er noch, ob die HTTP-Version nicht 1.0 oder 0.9 ist, da diese Versionen keine in Chunks aufgeteilten Nachrichten unterstützen.

Letztlich gibt es noch die Methoden *close()* und *flush()*. *Flush()* ruft die flush()-Methode des *BufferedOutputStreams* auf. Da dieser Stream ebenfalls im ChunkedOutputStream genutzt wird, ist es nicht nötig, die *flush()*-Methode von diesem extra aufzurufen.

```
1  public class ResponseOutputStream implements Flushable, Closeable, AutoCloseable {
2
3      @Override
4      public void close() throws IOException {
5          commitHeader();
6          flush();
7
8          if(!response.getHeaders().keepAlive()) {
9              if(out != null) {
10                 out.close();
11             }
12         }else {
13             if(chunkedOut != null) {
14                 chunkedOut.finish();
15             }
16         }
17     }
18 }
```

Listing 16: ResponseOutputStream close

Die *close()*-Methode, zu sehen in Listing 16, versucht als erstes die Headerdaten zu übermitteln. In Zeile 10 wird der *BufferedOutputStream* geschlossen, allerdings nur, wenn die Verbindung nicht geschlossen werden soll. Überprüft wird dies in Zeile 8, indem der *Keep-Alive*-Header überprüft wird. Wenn die Verbindung bestehen bleiben soll, darf der Stream natürlich nicht geschlossen werden. Stattdessen wird der ChunkedOutputStream beendet, indem seine *finish()*-Methode aufgerufen wird.

4.1.6 Cache

Wenn statische Ressourcen ausgeliefert werden, ist es sehr wahrscheinlich, dass dieselben Dateien immer wieder geöffnet und gelesen werden müssen. Das kann bei großer Serverlast zu sehr geringen Lesegeschwindigkeiten führen. Dafür wäre es praktisch, häufig genutzte Dateien im Arbeitsspeicher zu belassen, damit diese nicht neu gelesen werden müssen. Dafür sollte es eine Cache-Schnittstelle geben. Eine solche Schnittstelle sieht man in Listing 17.

```
1 public interface Cache<T, K> {
2
3     public void put(T key, K value);
4     public K get(T key);
5
6     public int size();
7
8     public K remove(T key);
9     public void clear();
10 }
```

Listing 17: Cache-Interface

Dieses Interface bietet implementierenden Klassen die grundlegenden Methoden zum Einfügen und Herausnehmen von Objekten. Da es sich um riesige Datenmengen handeln kann, birgt dies ohne weitere Vorkehrungen allerdings die Gefahr von OutOfMemory-Fehlern (OOM-Fehler). Um dies umgehen ist es allerdings erst einmal nötig zu verstehen, wie der Garbage-Collector von Java eigentlich funktioniert.

Wenn der Garbage-Collector die JVM pausiert, sucht er alle Objekte, auf die nicht mehr zugegriffen wird. Das heißt er sucht Objekte, auf die es keine Referenzen gibt.

```
1 String s = "Hello World";
2 s = "Also Hello World";
```

Listing 18: Beispiel Referenzen

In Listing 18 sieht man in Zeile 1 wie ein String-Objekt erstellt wird. In dem Moment gibt es eine Referenz auf das Objekt mit dem Inhalt *„Hello World“*. In Zeile 2 wird allerdings die Referenz überschrieben und das vorherige Objekt ist nicht mehr erreichbar. Der Garbage-Collector kümmert sich darum, dass der Arbeitsspeicher, der von dem vorherigen Objekt belegt wird, freigegeben wird. Das Objekt mit dem Inhalt *„Also Hello World“* wird nun referenziert. Wenn ein Objekt fest referenziert wird, nennt man es starke Referenz. Ein stark referenziertes Objekt wird niemals entfernt.

Es gibt allerdings noch weitere Arten von Referenzen: schwache-, weiche- und Phantomreferenzen. Der Vorteil dieser Referenzarten ist es, dass Objekte, auf die in dieser Art und Weise referenziert wird, unter bestimmten Voraussetzungen freigegeben werden können. Diese Referenzen werden durch die Java-Klassen *WeakReference*, *SoftReference* und *PhantomReference* dargestellt.
Im Falle eines Caches sind schwache und weiche Referenzen interessant. Der Garbage-Collector versucht, weiche Referenzen so lange wie möglich im Speicher zu halten. Falls er allerdings feststellt, dass ein OOM-Fehler verursacht werden würde, entfernt er auch weiche Referenzen. Schwache Referenzen werden beim nächsten Pausieren durch den Garbage-Collector entfernt, können bis dahin also noch genutzt werden. [17]
Durch diese zwei Referenzarten kann man also Objekte im Speicher behalten, die nicht unbedingt notwendig sind, aber doch Vorteile bringen können.

```
1 WeakReference<String> reference = new WeakReference<>("Hello World");
2
3 System.out.println(reference.get());
```

Listing 19: Beispiel WeakReference

In Listing 19 ist zu sehen, wie eine schwache Referenz erstellt wird. Dafür wird einfach ein Objekt vom Typ *WeakReference* erstellt, dem im Konstruktor ein Objekt übergeben wird. Um wieder an das Objekt zu kommen muss auf das Referenz-Objekt *get()* angewandt werden. Allerdings ist nicht garantiert, dass *get()* das ursprüngliche Objekt zurückgibt. Stattdessen kann auch *null* zurückgegeben werden. Dies bedeutet, dass das Objekt freigegeben wurde. Demnach kann in Zeile 3 die Ausschrift entweder *Hello World* oder *null* sein.
Unter diesen Gesichtspunkten wurde die Klasse *SimpleCache* implementiert. *SimpleCache* ist eine Implementierung des Cache-Interfaces, wodurch alle Methoden, die dort festgelegt wurden, implementiert werden müssen. Also unter anderem *put()* und *get()*. Diese Klasse benutzt, wie in Listing 20, Zeile 3 zu sehen, intern eine *ConcurrentHashMap,* um Schlüssel und Wertepaare einander zuweisen zu können. Eine *HashMap* in Java hat beim Einfügen, Entfernen und Herausnehmen meistens eine Laufzeit von O(1) [18], was diese perfekt für diese Art von Aufgabe macht. Da es sich außerdem um eine *ConcurrentHashMap* handelt, ist sie eine threadsichere Implementierung, wodurch mehrere Threads problemlos dasselbe Cache-Objekt nutzen können.

```
1  public class SimpleCache<K, V> implements Cache<K, V> {
2
3      final ConcurrentHashMap<K, CacheReference<K, V>> cacheMap =
4              new ConcurrentHashMap<K, CacheReference<K, V>>();
5      final ReferenceQueue<? super V> refQueue = new ReferenceQueue<>();
6
7      public SimpleCache() {
8          Thread t = new RemoveThread<K, V>(this);
9          t.setDaemon(true);
10         t.setName("CacheClearer");
11         t.start();
12     }
13
14     @Override
15     public void put(K key, V value) {
16         CacheReference<K, V> ref = new CacheReference<K, V>(value, (ReferenceQueue<? super V>) refQueue, key);
17         cacheMap.put(key, ref);
18     }
19
20     @Override
21     public V get(K key) {
22         CacheReference<K, V> ref = cacheMap.get(key);
23         if(ref == null)
24             return null;
25
26         return ref.get();
27     }
28
29   ...
30 }
```

Listing 20: Implementierung SimpleCache

In Zeile 5 wird eine *ReferenceQueue* erstellt. Eine *ReferenceQueue* ist eine Warteschlange, die an ein Referenzobjekt übergeben werden kann. Das Referenz-Objekt wird dann, sobald es freigegeben wurde, an die Queue angehangen. Dadurch ist es möglich herauszufinden, wann es freigegeben wurde.

In der *put()*-Methode wird in Zeile 16 eine *CacheReference*, eine selbst erstellte Klasse, die *SoftReference* erweitert, erstellt. Diese nimmt zusätzlich zu seinem Objekt, auf das referenziert wird, und einer Queue noch einen Schlüssel an. Dieser Schlüssel wird intern als starke Referenz gespeichert, da er nötig ist, um die Referenz in der HashMap wiederzufinden. Das referenzierte Objekt wird jedoch weiterhin wie ein weich referenziertes Objekt behandelt.

Das Referenz-Objekt wird dann in Zeile 17 als Wert, mit dem übergebenen Schlüssel, in die *HashMap* eingefügt.

In der *get()*-Methode wird in Zeile 22 die Referenz aus der *HashMap* geholt. Es wird dann verglichen, ob diese *null* ist und demnach entweder *null* oder das referenzierte Objekt zurückgegeben.

Im Konstruktor des *SimpleCaches* wird zusätzlich ein *RemoveThread* gestartet. Zu sehen ist dieser in Listing 21. Dieser wartet in Zeile 18 darauf, dass an die *ReferenceQueue* eine Referenz angehangen wird. Sobald eine Referenz angehangen wird, heißt das, dass das

referenzierte Objekt entfernt wurde. Folglich kann die Referenz aus dem Cache entfernt werden. Dies geschieht in Zeile 22.

```
1  final class RemoveThread<K, V> extends Thread {
2
3      private WeakReference<SimpleCache<K,V>> cache;
4
5      RemoveThread(SimpleCache<K, V> cache) {
6          this.cache = new WeakReference<>(cache);
7      }
8
9      @Override
10     public void run() {
11         while(true) {
12             SimpleCache<K,V> sCache = cache.get();
13
14             if(sCache == null)
15                 break;
16
17             try {
18                 CacheReference<K, V> ref = (CacheReference<K, V>) sCache.refQueue.remove(1000);
19                 if(ref != null) {
20                     ConcurrentHashMap<K, CacheReference<K, V>> map = sCache.cacheMap;
21                     if(map != null)
22                         map.remove(ref.key);
23                 }
24                 Thread.sleep(5000);
25             } catch (IllegalArgumentException | InterruptedException e) {
26             }
27         }
28     }
29 }
```

Listing 21: RemoveThread

Die Referenz des Caches wird als schwache Referenz gespeichert. Das sorgt dafür, dass das Cache-Objekt tatsächlich freigegeben werden kann. Sobald es freigegeben wurde, wird dies in der Schleife (in Zeile 14) erkannt. Dann wird die Schleife beendet und folgend auch der Thread. Somit wird der Thread beendet, sobald die letzte Referenz auf den Cache freigegeben wurde.

4.1.7 Container

Die Handler-Objekte sollen von einer zentralen Einheit festleg- und abrufbar sein. Folglich ist ein Container, der sich um das Handhaben der Handler kümmert, nötig. Dafür ist die in Listing 22 zu sehende abstrakte Klasse *HandlerContainer* gedacht. Der Container speichert statisch eine Instanz eines festgelegten Containers in Zeile 2 ab. Ein Container wird mit *setAsContainer*() gesetzt. Der Server kümmert sich darum, dass ein *HandlerContainer* gesetzt wird.

```java
public abstract class HandlerContainer {
    private static HandlerContainer container;

    public static void setAsContainer(HandlerContainer container) {
        HandlerContainer.container = container;
    }

    public static HandlerContainer getContainer() {
        if(container == null) {
            throw new NotImplementedException("No container set");
        }
        return container;
    }


    public abstract Handler getHandler(String name) throws NoHandlerException;
    public abstract void addHandler(Handler h) throws AlreadyHandledException;
}
```

Listing 22: HandlerContainer

Der gesetzte *HandlerContainer* kann dann mit *getContainer*() abgerufen werden. Falls kein Container gesetzt wurde, wird eine *NotImplementedException* geworfen. Dies sollte allerding nie auftreten, da sich der Server wie bereits beschrieben um die Implementation des Containers kümmert.

Zusätzlich bietet die Klasse die abstrakten Methoden *getHandler*() und *addHandler*(), mit denen Handler abgerufen und hinzugefügt werden können. Wie genau diese Methoden gehandhabt werden, wird in Abschnitt 4.2.4 erläutert.

So wie es einen Container für Handler gibt, gibt es auch einen Container für Clients. Diese *ClientContainer*-Klasse liegt ebenfalls in abstrakter Form in der API vor (Listing 23). Er funktioniert fast analog zum *HandlerContainer*, jedoch gibt es zusätzlich die *scheduleForExecution*()-Methode. Diese soll in der Implementierung einen Client annehmen und diesen dann asynchron, d.h. in einem separaten Thread ausführen.

```java
public abstract class ClientContainer {
    private static ClientContainer container;

    public static void setAsContainer(ClientContainer container) {
        ClientContainer.container = container;
    }

    public static ClientContainer getContainer() {
        if(container == null) {
            throw new NotImplementedException("No container set");
        }
        return container;
    }

    public abstract void scheduleForExecution(Client c);

    public abstract void addClient(Client c);
    public abstract Client getClient(String clientName);
}
```

Listing 23: ClientContainer

4.1.8 Hilfsklassen

HTTP hat viele Einzelheiten, für die es sich lohnen würde, eigene Klassen und Funktionalitäten anzubieten. Dazu gehören zum Beispiel das HTTP-Datum, Cookies, HTTP-Codes, Methoden, Ranges und vieles anderes. Dies ist nötig, um die Übersichtlichkeit und vereinfachte Funktionalität zu gewährleisten. Es wäre auch möglich, alle Header-Felder als String beizubehalten, dies würde allerdings vergleichende und verändernde Operationen unnötig verkomplizieren.

Dafür wurden vorläufig einige Klassen implementiert, die die Header-Feld-Werte als Text annehmen und diesen auswerten können. Diese Hilfsklassen umfassen aber bei weitem nicht alle Einzelheiten, da sie beliebig weit unterteilbar sind.

```
1  public class HttpDate implements Comparable<HttpDate>{
2
3      private Date date;
4
5      public HttpDate() {
6          setDate();
7      }
8
9      public HttpDate(String date) {
10         setDate(date);
11     }
12
13     ...
14 }
```

Listing 24: HttpDate Konstruktor

Die erste dieser Klassen ist *HttpDate*. Bei fast jeder Http-Anfrage oder Antwort werden Daten[1] gesendet. Diese können als Information, in Conditionals, zum Caching oder in Cookies verwendet werden. Dabei ist es oftmals nötig, Vergleiche mit diesen anzustellen. Dafür wurde die in Listing 24 zu sehende Klasse entwickelt.

Intern wird das Datum als normales Java-Date-Objekt gespeichert. Initialisiert wird es durch einen von zwei Konstruktoren. Der Konstruktor in Zeile 5 nimmt keine Argumente entgegen und setzt das Datum auf den aktuellen Zeitpunkt. Der in Zeile 9 nimmt einen String als Argument entgegen. Dieser String wird in der Methode *setDate()* verarbeitet.

[1] Hier als Plural von Datum

```
1  public void setDate(String date) {
2      if(date.endsWith(" GMT"))
3          date = date.substring(0, date.length() - 4);
4
5      if(date.charAt(3) == ',')           //Sun, 06 Nov 1994 08:49:37
6          this.date = dateStringToDate(date);
7      else if(date.charAt(3) == ' ')  //Sun Nov  6 08:49:37 1994
8          this.date = dateStringToDate(asciiToIMF(date));
9      else                            //Sunday, 06-Nov-94 08:49:37
10         this.date = dateStringToDate(rfc850ToIMF(date));
11 }
12
13 public void setDate(Date date) {
14     this.date = date;
15 }
16
17 public void setDate(long time) {
18     date = new Date(time);
19 }
20
21 public void setDate() {
22     date = new Date();
23 }
```

Listing 25: HttpDate setDate

Es gibt 4 *setDate()*-Methoden:

- eine die einen String annimmt und auswertet,
- eine die ein Datum annimmt und direkt speichert,
- eine die einen long-Wert annimmt und in ein Datum umwandelt und
- eine Methode, die ein neues Datum-Objekt mit der jetzigen Systemzeit erstellt.

Am interessantesten hierbei ist die Methode in Zeile 1. Diese akzeptiert sowohl das Datum im RFC-850-Format [19], im ASCII-Zeit-Format und im Internet Message Format (IMF). Das IMF ist die Variante, die in HTTP genutzt werden sollte, die anderen beiden Formate sind veraltet, werden aber noch akzeptiert.

Als erstes wird das GMT in Zeile 2 und 3 entfernt, sofern dies vorhanden ist. Danach wird in bestimmten Stellen des Datums-Strings geschaut, welche Zeichen vorhanden sind. Beispielsweise ist im IMF-Datum an der vierten Stelle immer ein Komma und an der vierten Stelle des ASCII-Datums immer ein Leerzeichen. Daraus lässt sich schließen, um welches Datums-Format es sich handelt. Es handelt sich hier allerdings eher um eine Heuristik, da Fremde oder fehlerhafte Formate zu falschen Interpretationen führen können. Für exakte Voraussagen müsste man durch reguläre Ausdrücke die Patterns vergleichen. Dies wäre allerdings ein viel höherer Aufwand, sowohl in der Implementation als auch bei der Laufzeit, der wenig Nutzen hat.

In Zeile 6 wird dann der Datums-String in ein Datums-Objekt umgewandelt, wenn es sich um IMF handelt. Wenn es sich um das ASCII-Format handelt, wird es in Zeile 8 als erstes in IMF umgewandelt und dann in ein Datum-Objekt. Wenn es keins von beiden ist wird es in das RFC-850-Format und dann in ein *Date* umgewandelt.

```
1    @Override
2    public String toString() {
3        SimpleDateFormat format = new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss", Locale.US);
4        format.setTimeZone(TimeZone.getTimeZone("GMT"));
5
6        return format.format(date) + " GMT";
7    }
```

Listing 26: HttpDate toString

Die Klasse bietet zusätzlich noch die Methoden *toString()*, *getDate()*, *equals()* und *compareTo()* an.

- *getDate()* gibt das Datum-Objekt zurück,
- *equals()* vergleicht die Daten-Objekte von 2 *HttpDates* auf Gleichheit und gibt wahr oder falsch zurück
- und *compareTo()* vergleicht die Daten mit derer *compareTo()*-Methode und gibt deren Ergebnis zurück.

Die *toString*-Methode, die in Listing 26 zu sehen ist, nimmt das Datum, formatiert es mit Hilfe eines *SimpleDateFormat*-Objektes in das IMF-Format und gibt das Ergebnis als String zurück.

Eine weitere dieser Klassen ist die *Cookie*-Klasse, zu sehen in Listing 27. Cookies können sowohl in Anfragen als auch in Antworten verwendet werden, dabei aber jeweils in verschiedenen Headern. Im Cookie-Request-Header werden diese als Wert-Schlüssel-Paare übersendet und im Set-Cookie-Response-Header können weitere Information hinzugefügt werden.

```
1  public class Cookie {
2
3      private String name;
4      private String value;
5
6      public Cookie() { }
7
8      public Cookie(String cookie) {
9          String[] splits = cookie.trim().split("=");
10
11         name = splits[0];
12         value = splits[1];
13     }
14
15     public void setName(String name) { this.name = name; }
16
17     public void setValue(String value) { this.value = value; }
18
19     public String getName() { return name; }
20
21     public String getValue() { return value; }
22
23   ...
24 }
```

Listing 27: Cookie

Da die *Cookie*-Klasse für beide Fälle verwendet werden soll, ist es nötig einen Konstruktor anzubieten, der Werte-Paare auswertet (Zeile 8) und einen der keine weiteren Einstellungen vornimmt (Zeile 6).

Da in Requests Schlüssel und Wert durch ein Gleichheitszeichen getrennt werden, wird der Cookie-String bei diesem aufgeteilt (Zeile 9). Aber erst nachdem vorlaufende und nachfolgende Leerzeichen durch die *trim()*-Methode entfernt wurden. Im ersten Teil befindet sich der Name des Wertes, im zweiten der Wert. Diese werden in Zeile 11 und 12 in die klassenweiten Variablen, *name* und *value*, abgespeichert.

Diese Werte können ebenfalls durch einen Aufruf der Methoden *setName()* und *setValue()* festgelegt werden. Abrufen kann man diese durch die Methoden *getName()* und *getValue()*. Durch diese Methoden ist bereits das Erstellen von Cookies bei Requests gedeckt. In Antworten werden allerdings noch weitere Methoden benötigt.

4.2 Server

Der Server kümmert sich um die Verwaltung von Modulen, das Erstellen von Ordnern und um das Bereitstellen von Containern für Handler und Clients. Begonnen wird wie in jedem Java-Programm mit der *main*()-Methode. Diese befindet sich in der *ServerStart*-Klasse. Die *main*()-Methode ist in Listing 28 zu sehen.

```
1  public class ServerStart {
2
3      public static void main(String[] args) throws Exception {
4          System.out.println("starting server ...");
5          System.out.println("setting containers");
6
7          HandlerContainer.setAsContainer(new WebDragonHandlerContainer());
8          ClientContainer.setAsContainer(new WebDragonClientContainer());
9
10         System.out.println("creating folders");
11
12         try {
13             new FileCreator().create();
14         } catch (IOException e) {
15             System.out.println("Error while creating necessary files, shutting down now.");
16             System.exit(-1);
17         }
18
19         new HandlerIntializer().initialize();
20     }
21 }
```

Listing 28: ServerStart - die main()-Methode

In Zeile 7 und 8 werden als erstes die Handler- und ClientContainer Container gesetzt. Dies erfolgt in beiden Fällen mit der *setAsContainer*()-Methode. Danach werden in Zeile 13 alle nötigen Ordner erstellt. Falls etwas dabei fehlschlägt wirft die *create*()-Methode der *FileCreator*-Klasse eine *IOException*. In dem Fall, dass diese Exception geworfen wird, kann nicht fortgefahren werden und der Server wird beendet. Wenn es allerdings erfolgreich war, werden alle Module, Handler, Listener und Clients geladen, indem die *initialize*()-Methode der *HandlerInitializer*-Klasse aufgerufen wird.

Sobald die *main*() durchgelaufen ist, wurde entweder mindestens ein *Listener* gestartet, der das Programm am Laufen hält, oder wenn kein weiterer Thread gestartet wurde, wird der Server sofort beendet. Der Server ist also entweder einsatzbereit, oder wäre nutzlos, wenn es keine Module mit Listenern gibt.

4.2.1 Erstellen nötiger Ordner und Dateien

Der Server erstellt beim Starten alle nötigen Ordner, falls sie noch nicht existieren. Dabei ist es wichtig, diese an einer geeigneten Position im Filesystem anzulegen. Es gibt 3 wichtige Ordner:

- Den *modules* Ordner, der Jar-Dateien mit Modulen enthält,
- Den *lib* Ordner, der externe Bibliotheken enthält, die von mehreren Modulen und vom Server genutzt werden können (beispielsweise die API-Jar) und
- Den *config* Ordner, in dem die Konfigurationsdaten der Module gespeichert werden.

Um das Erstellen diese Ordner kümmert sich die Klasse *FileCreator,* die in Listing 29 zu sehen ist. Dies geschieht beim Aufruf der *create*()-Methode. Als erstes wird in dieser die *setDirectoryPaths*()-Methode aufgerufen. Diese erfragt in Zeile 13 durch Aufruf der statischen Methode *getCurrentPath*() den Ordner in dem sich die Serversoftware befindet. Danach werden *File*-Objekte erstellt, die ausgehend vom *Server*-Ordner, die Unterordner bezeichnen. Als nächstes wird in der *create*()-Methode *makeDirectories*() aufgerufen. Diese Methode erstellt alle 3 Unterordner.

```
1  public class FileCreator {
2
3      private static File modules;
4      private static File config;
5      private static File lib;
6
7      public void create() throws IOException {
8          setDirectoryPaths();
9          makeDirectories();
10     }
11
12     private void setDirectoryPaths() {
13         String path = getCurrentPath();
14
15         modules = new File(path + File.separator + "modules");
16         config = new File(path + File.separator + "config");
17         lib = new File(path + File.separator + "lib");
18     }
19
20     private void makeDirectories() {
21         modules.mkdirs();
22         config.mkdirs();
23         lib.mkdirs();
24     }
25
26     public static String getCurrentPath() {...}
27 }
```

Listing 29: FileCreator - Erstellen von Ordnern

Zusätzlich kümmert sich die *FileCreator*-Klasse noch um das Erstellen von Konfigurationsdateien. Das erledigt sie, indem die *createConfigFile*()-Methode, die in Listing 30 zu sehen ist, aufgerufen wird. Als Übergabeparameter wird eine Liste von Modulen erwartet. Für jedes dieser Module wird in Zeile 5 ein *File*-Objekt erstellt, das als Dateinamen den Namen des Moduls und die Dateiendung *conf* nutzt. Zu bemerken ist, dass dies die Datei nicht selbst erstellt, sondern das Erstellen die Aufgabe des Moduls ist.

Nachdem das File-Objekt erstellt wurde wird *createConfigFile*() und danach *loadConfigFile*() aufgerufen.

```
1  public class FileCreator {
2
3      public void createConfigFile(List<Module> modules) throws IOException {
4          for(Module module : modules) {
5              File configFile = new File(getConfigDirectory() +
6                                         File.separator + module.getModuleName() + ".conf");
7
8              module.createConfigFile(configFile);
9              module.loadConfigFile(configFile);
10         }
11     }
12
13 }
```

Listing 30: FileCreator - Erstellen von Konfigurationsdateien

Die Klasse bietet außerdem Getter-Methoden, die die erstellten *File*-Objekte der Ordner zurückgeben.

4.2.2 Laden von Modulen

Module sollen im *modules*-Ordner, der beim Starten des Servers erstellt wird, in Jar-Dateien abgelegt werden. Diese Module werden beim Serverstart von der *ModuleLoader*-Klasse geladen. Dafür ist es als erstes nötig, alle Jar-Dateien aus diesem Ordner und allen Unterordnern zu finden. Dies wird von der Klasse *JarFileLoader* übernommen, die in Listing 31 zu sehen ist.

```
1  class JarFileLoader {
2
3      public List<File> getJarFilesInDirectory(File directory) {
4          List<File> jars = new LinkedList<>();
5
6          File[] files = directory.listFiles();
7
8          for(File f : files) {
9              if(f.isDirectory()) {
10                 jars.addAll(getJarFilesInDirectory(f));
11             } else {
12                 try {
13                     JarFile jar = new JarFile(f);
14                     jar.close();
15
16                     jars.add(f);
17                 } catch (IOException e) {
18                     //do nothing if not jar
19                 }
20             }
21         }
22
23         return jars;
24     }
25 }
```

Listing 31: JarFileLoader

Sie besitzt lediglich die Methode *getJarFilesInDirectory*(). Diese nimmt als Übergabeparameter den Ordner, der nach Jars durchsucht werden soll, an. Als erstes wird in Zeile 4 eine Liste erstellt, die alle gefundenen Jars beinhaltet. Danach werden alle sich im Ordner befindenden Dateien in einem *File*-Array aufgelistet. In Zeile 8 beginnt eine Schleife, die über jede dieser Dateien iteriert. Dabei wird als erstes geschaut, ob es sich um einen Ordner handelt. Ist es ein Ordner, wird die Methode *getJarFilesInDirectory*() rekursiv aufgerufen und das Ergebnis in die *jars*-Liste eingefügt. Wenn es eine Datei ist, wird versucht, ein *JarFile*-Objekt (in Zeile 13) aus der Datei zu erstellen. Wenn es sich dabei nicht um eine Jar handelt, wird eine *IOException* geworfen und die Datei ignoriert. Ist es jedoch eine Jar, wird diese als erstes geschlossen (da das Erstellen eines *JarFile*-

Objekts diese Jar öffnet) und folgend wird die Datei in die *jars*-Liste eingefügt. Abschließend wird die Liste der Jars zurückgegeben.

```
1  public class ModuleLoader {
2
3      private static final HashMap<String, Module> modules = new HashMap<>();
4
5      public static void addModule(Module module) {...}
6
7      protected static List<Module> getModules() {...}
8
9      protected static void loadModules() {...}
10
11     private static Module makeClass(Class<?> cls) {...}
12
13 }
```

Listing 32: ModuleLoader

Die Klasse *ModuleLoader*, die in Listing 32 zu sehen ist, kümmert sich um das Laden und Zwischenspeichern von Modulen. Sie besitzt eine *HashMap*, die Strings, genauer die Namen von Modulen, als Schlüssel verwendet und Module zu diesem Schlüssel zuordnet. Module werden durch einen Aufruf der *addModule*()-Methode in die *Map* eingefügt. Alle bisher eingefügten Module werden durch einen Aufruf der *getModules*()-Methode abgefragt. Zum Laden der Module sind allerdings die Methoden *loadModules*() und *makeClass*() wichtig. Die Methode *loadModules*() ist in Listing 33 zu sehen. Als erstes werden durch Aufruf der *getJarFilesInDirectory*()-Methode alle Jars abgefragt. Danach wird über jede Jar iteriert. in jeder Iteration wird in Zeile 4 ein *JarFile*-Objekt erstellt. Von diesem wird jeder Eintrag als *Enumeration* abgefragt. In Zeile 7 und 8 wird ein *URLClassLoader* erstellt. Dieser *ClassLoader* ist in der Lage, *class* Dateien dynamisch einzulesen.

```
1  List<File> jars = new JarFileLoader().getJarFilesInDirectory(FileCreator.getModuleDirectory());
2
3  for(File jar : jars) {
4      try (JarFile jarFile = new JarFile(jar)) {
5          Enumeration<JarEntry> e = jarFile.entries();
6
7          URL[] urls = { new URL("jar:file:" + jar + "!/") };
8          URLClassLoader cl = URLClassLoader.newInstance(urls);
9
10         while (e.hasMoreElements()) {
11             JarEntry je = e.nextElement();
12             if(je.isDirectory() || !je.getName().endsWith(".class")){
13                 continue;
14             }
15             // -6 because of .class
16             String className = je.getName().substring(0,je.getName().length()-6);
17             className = className.replace('/', '.');
18             Class<?> c = cl.loadClass(className);
19
20             Module m = makeClass(c);
21
22             if(m != null) {
23                 addModule(m);
24             }
25         }
26     } catch(IOException e) {
27         e.printStackTrace();
28     } catch(ClassNotFoundException e) {
29         e.printStackTrace();
30     }
31 }
```

Listing 33: loadModules()-Methode

Ab Zeile 10 wird über jeden Eintrag der Jar iteriert. Dabei wird in Zeile 12 überprüft, ob es sich bei diesem Eintrag um einen Ordner handelt oder es eine Datei ist, die keine Klasse ist. Wenn diese Konditionen wahr sind, wird zur nächsten Schleifeniteration gesprungen, ansonsten wird normal fortgefahren. Nach der Überprüfung ist es sicher, dass es sich um eine *class*-Datei handelt. Von dieser wird nun der Klassenname ermittelt, indem der komplette Name abgefragt und durch die letzten 6 Zeichen (auf Grund der .class Dateiendung) abgeschnitten werden. Danach wird der Pfadseparator Slash durch einen Punkt ersetzt, da Klassennamen durch Punkte von Paketnamen getrennt werden.
In Zeile 18 wird nun versucht die Klasse mit Hilfe des in Zeile 8 erstellten *ClassLoaders* zu laden. Danach wird ein *Module* durch den Aufruf der *makeClass*()-Methode erstellt. Falls es sich nicht um eine *Module*-Klasse handelt, wird von dieser *null* zurückgegeben. Letztlich wird das Modul durch *addModule*() hinzugefügt und in die nächste Iteration übergegangen.
Die *makeClass*()-Methode ist in Listing 34 zu sehen. Diese ist nicht sehr komplex. Es wird als erstes in Zeile 1 überprüft, ob es sich um eine Klasse handelt. Die *isAssignableFrom*()-Methode der Klasse *Class* überprüft, ob diese Klasse eine erbende Klasse von Module ist.

```
1 if (Module.class.isAssignableFrom(cls)) {
2     System.out.println(cls.getName() + " found");
3     try {
4         return (Module) cls.getDeclaredConstructor().newInstance();
5     } catch (InstantiationException | IllegalAccessException | IllegalArgumentException
6             | InvocationTargetException | NoSuchMethodException | SecurityException e) {
7
8         return null;
9     }
10 } else {
11     return null;
12 }
```

Listing 34: makeClass()-Methode

Wenn es sich um eine Unterklasse handelt, wird in Zeile 4 der Konstruktor, der keine Argumente annimmt, von dieser geholt und eine neue Instanz mit der Hilfe von diesem Konstruktor erstellt. Durch Reflection eine Instanz einer Klasse zu erstellen, kann sehr fehleranfällig sein, wie durch die hohe Anzahl von möglichen Exceptions im *catch*-Block in Zeile 5 und 6 zu sehen ist. Wenn es zu einer *Exception* kommt oder die Klasse keine Unterklasse von *Module* ist, wird *null* zurückgegeben, die Klasse wird also ignoriert.

Durch diesen Vorgang können nun alle Module, die sich im *modules*-Ordner befinden, geladen werden.

4.2.3 Laden von Handlern, Listenern und Clients

Nachdem die Module geladen wurden, müssen noch die Handler, Listener und Clients geladen werden. Für diesen Zweck gibt es die Klassen *HandlerLoader, ListenerLoader und ClientLoader*. Diese Klassen funktionieren analog, weswegen nur die Klasse *HandlerLoader* beschrieben wird. Sie besitzt die Methoden *loadHandler*() und *makeObject*(). Die Methode *loadHandler*() ist in Listing 35 zu sehen.
Sie erwartet als Übergabeparameter eine Liste von Klassen, die Handler erweitern und gibt eine Liste von Handlern zurück. Die Handler-Klassen lässt man sich von einem Modul geben, indem man dessen Methode *getHandler*() aufruft[1].
Es wird über jede Klasse iteriert und in Zeile 9 ein Handler-Objekt mit Hilfe der *makeObject*()-Methode erstellt. Dieses wird dann in die *loadedHandlers*-Liste hinzugefügt, solange es erzeugt werden konnte. Zum Schluss wird die Liste zurückgegeben.

[1] Beschrieben in Abschnitt 4.1.3 Module

```java
public class HandlerLoader {

    protected static List<Handler> loadHandler
      (List<Class<? extends Handler>> handlers) {

        List<Handler> loadedHandlers = new LinkedList<>();

        for(Class<? extends Handler> handler : handlers) {
            Handler loaded = makeObject(handler);

            if(loaded != null) {
                loadedHandlers.add(loaded);
            }
        }
        return loadedHandlers;
    }
   ...
}
```

Listing 35: loadHandler()

Die *makeObject*()-Methode macht nichts weiter als zu versuchen, eine neue Instanz eines Handler-Objekts durch Reflection zu erzeugen. Falls dies nicht möglich ist, gibt sie null zurück.

4.2.4 Der HandlerContainer

Handler, die geladen werden, werden in einem zentralen Container festgehalten. Eine Abstraktion des Containers ist bereits in der API vorhanden (Abschnitt 4.1.7). Die genaue Implementierung ist dabei allerdings noch offengeblieben.

```java
public class WebDragonHandlerContainer extends HandlerContainer {

    private final static HashMap<String, Handler> registeredHandlers = new HashMap<>();

    @Override
    public Handler getHandler(String name) throws NoHandlerException{
        Handler handler = registeredHandlers.get(name);

        if(handler == null)
            throw new NoHandlerException();

        return handler.getInstance();
    }

    @Override
    public void addHandler(Handler h) throws AlreadyHandledException {
        if(registeredHandlers.containsKey(h.getHandlerName()))
            throw new AlreadyHandledException("the handler " + h.getHandlerName() + " already exists");

        registeredHandlers.put(h.getHandlerName(), h);
    }

}
```

Listing 36: Implementierung HandlerContainer

Diese Implementierung ist in Listing 36 zu sehen. Registrierte *Handler* werden in einer *HashMap* gespeichert. Als Schlüssel wird hierbei der Name der Handler verwendet. Implementiert werden die Methoden *getHandler*() und *addHandler*(). Die Methode *getHandler*() versucht in Zeile 7, einen *Handler* mit dem übergebenen Namen aus der *HashMap* zu holen. Wenn es keinen *Handler* mit diesen Namen gibt, wird eine *NoHandlerException* geworfen. Falls es allerdings einen Handler gibt, wird eine neue Instanz dieses *Handlers* erstellt und diese zurückgegeben. Der Aufrufer kann diesen *Handler* nun verwenden.

In der addHandler()-Methode wird ein *Handler* als Übergabeparameter erwartet. Danach wird geschaut, ob es bereits einen *Handler* mit diesem Namen gibt. Wenn ja wird eine *AlreadyHandledException* geworfen. Ansonsten wird der *Handler* in die *HashMap* mit seinen Namen als Schlüssel eingefügt.

4.2.5 Der ClientContainer

Clients sollen alle ebenfalls über einen zentralen Container verwaltet werden. Dafür ist der *ClientContainer* gedacht. Dieser befindet sich, genau wie der *HandlerContainer,* bereits in abstrakter Form in der API. Es muss nur noch die *addClient*()-Methode implementiert werden. Diese wurde in der Klasse, die in Listing 37 zu sehen ist, implementiert.

```
1  public class WebDragonClientContainer extends ClientContainer {
2
3      private static ThreadPoolExecutor threadPool =
4        (ThreadPoolExecutor) Executors.newCachedThreadPool();
5
6      public void scheduleForExecution(Client c) {
7          threadPool.execute(c);
8      }
9
10     public int getThreadPoolSize() {
11         return threadPool.getPoolSize();
12     }
13 }
```

Listing 37: Implementierung ClientContainer

In Zeile 3 kann man sehen, dass die Klasse einen *ThreadPoolExecutor* beinhaltet. Einem *ThreadPoolExecutor* kann man einfache *Runnables* übergeben und diese werden von diesem Pool ausgeführt. Dies bietet einige Vorteile gegenüber dem Erstellen von *Threads* für jede *Runnable*. Jeder *Thread* benötigt seinen eigenen Speicher und muss vom

Betriebssystem verwaltet werden[1]. Das führt zu hohen Overheads beim Erstellen von Threads. Beim Verwenden von *ThreadPools* werden bereits vorhandene Thread benutzt und keinen neuen erstellt, solange es nicht notwendig ist, was den Overhead stark reduziert.
Die Methode *scheduleForExecution*() nimmt einen Client als Parameter an und übergibt diesen an den *ThreadPool*. Durch *getThreadPoolSize*() kann abgefragt werden, wie viele *Threads* von dem im Container verwendeten Pool gerade aktiv sind. Da dies keine Methode ist, die von der abstrakten Klasse geerbt wird, ist diese nur dem Server selbst bekannt. Sie wird nur zu Debug- und Testzwecken benötigt.

Natürlich besitzt auch diese Implementierung die *addClient*()- und *getClient*()-Methoden (Listing 38). Die Clients werden durch Aufruf der *addClient*()-Methode in die *HashMap registeredClients* eingefügt. Wenn es bereits einen Client mit dem Namen gibt, wird eine *AlreadyHandledException* geworfen. Abgefragt wird ein Client durch *getClient*(), dafür wird der Name des gewünschten *Clients* als Parameter übergeben. Falls kein *Client* mit diesem Namen existiert, wird eine *NotImplementedException* geworfen. Ansonsten wird eine neue Instanz des Clients erstellt und zurückgegeben, indem seine *getInstance*()-Methode aufgerufen wird.

```
1 public class WebDragonClientContainer extends ClientContainer {
2     private final static HashMap<String, Client> registeredClients = new HashMap<>();
3
4     public void addClient(Client c) {
5         if(registeredClients.containsKey(c.getName()))
6             throw new AlreadyHandledException();
7
8         registeredClients.put(c.getName(), c);
9     }
10
11    public Client getClient(String clientName) {
12        Client client = registeredClients.get(clientName);
13
14        if(client == null)
15            throw new NotImplementedException();
16
17        return client.getInstance();
18    }
19 }
```

Listing 38: Implementierung ClientContainer add() und get()

[1] Sofern das Betriebssystem dies unterstützt, ansonsten verwaltet die JVM die Threads.

4.2.6 Komplette Handler-Initialisierung

Nachdem nun alle Module, Handler, Listener und Clients geladen werden können, ist es nötig, diese gesamten Funktionalitäten aufzurufen. Dies findet in der Klasse *HandlerIntializer* statt, genauer genommen durch die Methode *initialize*(), die in Listing 39 zu sehen ist. Als erstes werden in Zeile 5 alle Module aus dem *modules*-Ordner geladen. Danach wird in Zeile 6 das Standardmodul geladen.

```
1  public class HandlerIntializer {
2
3      public void initialize() throws IOException {
4          System.out.println("loading Modules ...");
5          ModuleLoader.loadModules(FileCreator.getModuleDirectory());
6          ModuleLoader.addModule(new DefaultModule());
7
8          new FileCreator().createConfigFile(ModuleLoader.getModules());
9
10         List<Module> modules = ModuleLoader.getModules();
11
12         for(Module module : modules) {
13             registerHandlers(module.getHandler());
14             registerListeners(module.getListeners());
15             registerClients(module.getClients());
16         }
17     }
18 }
```

Listing 39: HandlerInitializer initialize()

Da sich dieses direkt im Servercode befindet, kann es einfach durch das Schlüsselwort *new* geladen werden. Nachdem die Module alle geladen wurden, werden die Konfigurationsdateien erstellt und geladen. Dies wird in der Methode *createConfigFile*() der Klasse *FileCreator* gehandhabt.

Zum Abschluss werden noch alle Handler und Listener in den Zeilen 12-16 geladen und registriert. Dies geschieht durch die Methoden *registerHandlers*(), *registerListeners*() und *registerClients*().

Zu sehen sind diese Methoden in Listing 40. Die *registerHandlers*()-Methode lädt als erstes alle Handler und fügt diese in den *HandlerContainer* ein, indem die *addHandler*() Methode aufgerufen wird. Die *registerListeners*()- und die *registerClients*()-Methode funktionieren ähnlich, nur dass die Listener durch den Aufruf der *registerListener*()-Methode der Klasse *ServerListener* und *Clients* durch den Aufruf der *addClient*()-Methode der Klasse ClientContainer *registriert* werden.

```java
public class HandlerIntializer {

    private void registerHandlers(List<Class<? extends Handler>> handlerClasses) {
        List<Handler> handlers = HandlerLoader.loadHandler(handlerClasses);

        for(Handler handler : handlers) {
            System.out.println("Handler: " + handler.getHandlerName() + " found");
            HandlerContainer.getContainer().addHandler(handler);
        }
    }

    private void registerListeners(List<Class<? extends Listener>> listenerClasses) {
        List<Listener> listeners = ListenerLoader.loadListener(listenerClasses);

        for(Listener listener : listeners) {
            System.out.println("Listener: " + listener.getClass() + " found");
            ServerListener.getInstance().registerListener(listener);
        }
    }

    private void registerClients(List<Class<? extends Client>> clientClasses) {...}
}
```

Listing 40: HandlerInitializer register() Methoden

4.2.7 Registrieren von Listenern – die ServerListener-Klasse

Listener, die geladen werden, sollen in diesem Moment auch beginnen Verbindungen anzunehmen und diese zu verarbeiten. Dafür muss deren *listen*()-Methode aufgerufen werden. Da es sich dabei um blockierende und eventuell unendlich wiederholende Aktionen handeln kann, sollte dies in einem zusätzlichen Thread ausgeführt werden. Darum und um weitere Funktionen kümmert sich die Klasse *ServerListener* (Listing 41). Bei dieser Klasse handelt es sich um ein Singleton, das heißt es gibt nur eine Instanz dieser Klasse. Diese wird durch die *getInstance*()-Methode angefragt.

```
1  public class ServerListener {
2
3      private List<Listener> listenerList = new ArrayList<Listener>();
4
5      private void startListening(Listener listener) {
6          Thread listeningThread = new Thread(new Runnable() {
7              @Override
8              public void run() {
9                  listener.listen();
10             }
11         });
12
13         listeningThread.start();
14     }
15
16     public void registerListener(Listener listener) {
17         listenerList.add(listener);
18
19         startListening(listener);
20     }
21
22     public List<Listener> getListeners() {...}
23 }
```

Listing 41: ServerListener

Registriert werden Listener durch die *registerListener*()-Methode, die in Zeile 16 zu sehen ist. Diese fügt den *Listener* als erstes zu einer Liste hinzu und ruft danach die Methode *startListening*() auf. In dieser wird ein neuer Thread erstellt, der nur die *listen*()-Methode ausführt. Zusätzlich gibt es noch die Methode *getListeners*(), die alle registrierten Listener zurückgibt.

4.2.8 Starten des Servers

Zum Starten des Servers muss in den Ordner gewechselt werden, in dem Webdragon.jar liegt. In diesem muss man den Befehl:

„java -cp lib/*:Webdragon.jar server.maettel.start.ServerStart“

ausführen.

Das Argument „-cp“ legt den Class-Path fest, die Pfade in denen Jars und Klassen liegen. Hier wird also die Webdragon.jar und alle Klassen und Jars die im lib-Ordner liegen dem Class-Path hinzugefügt. Das Argument „server.maettel.start.ServerStart“ zeigt, welche Klasse die *main*()-Methode beinhaltet von der aus gestartet werden soll.

4.3 Das Standard-Modul

Es soll bei dem Server standardmäßig ein Modul, welches HTTP/1.1 Funktionalität bietet, integriert sein. Dieses Modul wird durch die Klasse *DefaultModule*, zu sehen in Listing 42, dargestellt. Es ist anders, als andere Module fest im Serverquellcode eingebaut. Folglich muss es also nicht durch Reflection geladen werden.

```
1  public class DefaultModule extends Module {
2
3      @Override
4      public List<Class<? extends Handler>> getHandler() {
5          List<Class<? extends Handler>> handlers = new LinkedList<>();
6
7          handlers.add(DefaultHttpHandler.class);
8          handlers.add(DefaultFileHandler.class);
9
10         return handlers;
11     }
12
13     @Override
14     public List<Class<? extends Listener>> getListeners() {
15         List<Class<? extends Listener>> listeners = new LinkedList<>();
16
17         listeners.add(HttpListener.class);
18
19         return listeners;
20     }
21
22     @Override
23     public String getModuleName() {
24         return "DefaultHTTPModule";
25     }
26
27     ...
28 }
```

Listing 42: DefaultModule, Listener und Handler

Wie in Zeile 1 zu sehen ist, erweitert die Klasse die abstrakte Klasse *Module*. Es werden also alle abstrakten Methoden von v*Module* implementiert. Zu dem Modul gehören die Handler *DefaultHttpHandler* und *DefaultFileHandler*. Diese werden in der Methode *getHandler*(), zu sehen in Zeile 4 bis Zeile 11 zurückgegeben. Zusätzlich gehört zu dem Modul der Listener *HttpListener*, welcher in der Methode getListeners() zurückgegeben wird. Als Name des Moduls wurde *DefaultHTTPModule* festgelegt, der in der Methode *getModuleName*() zurückgegeben wird.

```
1  public class DefaultModule extends Module {
2
3      ...
4
5      public void createConfigFile(File configFile) throws IOException {
6          if(configFile.exists()) {
7              return;
8          }
9
10         JSONObject main = new JSONObject();
11         JSONObject host = new JSONObject();
12         host.put("hostname", "/path/to/www");
13
14         main.put("httpPort", 8081);
15         main.put("defaultHost", "/path/to/www");
16         main.put("hosts", host);
17
18         try (FileWriter fw = new FileWriter(configFile)) {
19             fw.write(main.toJSONString());
20             fw.flush();
21         }
22     }
23
24     public void loadConfigFile(File configFile) throws IOException{
25         try {
26             OptionLoader.loadOptions(configFile, this);
27         } catch (ParseException | URISyntaxException e) {
28             throw new IOException("could not load configFile", e);
29         }
30     }
31 }
```

Listing 43: DefaultModule Erstellung der Einstellungsdatei

Ein wichtiger Punkt ist noch das Erstellen einer Konfigurationsdatei. In dieser soll einstellbar sein:

- an welchem Port der Server auf HTTP-Verbindungen lauscht,
- an welchem Speicherort sich die Dateien befinden, die der Server anbietet und
- welche Hosts der Server verwaltet, zusammen mit ihren Speicherort.

Eine Datei mit diesen Einstellungsmöglichkeiten wird in der Methode *createConfigFile*() (Zeile 5, Listing 43) erstellt. Diese Datei wird im JSON-Format erstellt, da diese relativ einfach zu erstellen, zu ändern und einzulesen ist. Dafür wurde die Open-Source-Bibliothek json-simple [30] genutzt.

In der *createConfigFile*()-Methode wird als erstes überprüft, ob die Datei bereits existiert. Wenn sie bereits vorhanden ist, wird das Erstellen beendet, da sonst die komplette Datei überschrieben werden würde und somit das Ändern von Einstellungen nicht möglich wäre. Danach werden in den Zeilen 10 und 11 die zwei JSON-Objekte *main* und *host*

erstellt. Das *main*-Objekt ist vergleichbar mit einem Container, zu dem alle Einstellungsfelder eingefügt werden. Das *host*-Objekt ist ein Container, in dem alle Hosts, den der Server verwaltet, eingetragen werden.

Dafür wird in Zeile 12 als Platzhalter *hostname*, mit dem Pfad "/path/to/www" in host eingefügt. Diese müssen später nur angepasst werden, um festgelegte Hosts anzubieten. Danach wird (Zeile 14-16):

- als *httpPort* der Wert 8081,
- als *defaultHost* der Pfad "/path/to/www" und
- das *host*-Objekt in main eingefügt.

In diesem Fall wird *defaultHost* dafür verwendet, einen Standardpfad festzulegen für den Fall, dass ein unbekannter Host angefragt wurde.

Letztlich werden in Zeile 18 bis 20 die JSON-Objekte in die Einstellungsdatei ausgeschrieben.

```
{
    "httpPort":8081,
    "defaultHost":"/path/to/www",
    "hosts":
    {
        "hostname":"/path/to/www",
    }
}
```

Abbildung 9: Standard Konfigurationsdatei

In Abbildung 9 kann man nun sehen, wie die entstandene Datei aussieht. In dieser kann man nun die Werte so anpassen, dass sie zu dem eigenen Serverumfeld passen. In *hosts* kann man zusätzlich mehrere zusätzliche Hosts einfügen. Eine fertig ausgefüllte Konfigurationsdatei sieht man in Abbildung 10.

```
{
    "httpPort":8081,
    "defaultHost":"/home/maettel/httpServer/www",
    "hosts":
    {
        "maettel.de:8081":"/home/maettel/httpServer/wwwhost",
        "reloaded-server.de:8081":"/home/maettel/httpServer/wwwhost",
    }
}
```

Abbildung 10: Ausgefüllte Konfigurationsdatei

Eingelesen wird diese dann in Listing 43, Zeile 24. Darin wird eine statische Methode der Klasse *OptionLoader* aufgerufen, die die Konfigurationsdatei und eine Referenz auf ein Modul annimmt. In dieser wird die JSON-Datei analysiert und alle Einstellungen in die Options-Klasse eingespeist.

4.3.1 Der HttpListener

Nachdem der Server sich um das Erstellen und Laden von Konfigurationen gekümmert hat, lädt er als erstes alle *Listener*- und *Handler*-Klassen. Sobald dies erledigt ist, werden die *Listener* initialisiert. Das Standardmodul beinhaltet zurzeit nur einen *Listener*, den *HttpListener*. Dieser kümmert sich um das Annehmen und Bearbeiten von Verbindungen auf dem in den Konfigurationsdateien eingestellten Port. Den Handler kann man in Listing 44 sehen.

```
1  public class HttpListener implements Listener {
2
3      @Override
4      public void listen() {
5          int port =  Math.toIntExact((long) Options.getOption("httpPort", new DefaultModule()));
6
7          try (final ServerSocket ss = new ServerSocket(port)) {
8              while (true) {
9                  Socket s = ss.accept();
10
11                 Client client = ClientContainer.getContainer().getClient("Http_client");
12                 client.setSocket(s);
13                 ClientContainer.getContainer().scheduleForExecution(client);
14             }
15         } catch (IOException e) {
16             e.printStackTrace();
17         }
18     }
19 }
```

Listing 44: HttpListener

Da es sich bei dieser Klasse um einen *Listener* handelt, ist es natürlich nötig, dass die *listen*()-Methode implementiert wird. Diese wird von dem Interface *Listener* vorgeschrieben. In dieser Methode wird als erstes der festgelegte Port abgefragt (Zeile 5). Dafür wird die *Options*-Klasse, welche in der API vorhanden ist, benutzt. Die statische Methode *getOption*() nimmt hierbei als Argument einen String entgegen, welcher die Option beschreibt, und ein Module, was dafür sorgt, dass nur die Einstellungen dieses Modules benutzt werden. Da *getOption*() ein *Object* zurückgibt, muss dieses gecastet werden. Es wird als erstes in ein *long* gecastet, da json-simple 64-bit Datentypen für Zahlen benutzt. In einen Integer wird es durch die *toIntExact*()-Methode von *Math* gecastet.

Da nun der Port bekannt ist, kann der *ServerSocket* erstellt werden. Ein *ServerSocket* nimmt Verbindungen an und erstellt für jede eingehende Verbindung einen *Socket*, der zur Kommunikation benutzt werden kann. Erstellt wird dieser in Zeile 7, im *try*-Block. Danach werden in einer Endlosschleife alle eingehenden Verbindungen durch die *ServerSocket*-Methode *accept*() angenommen. Als nächstes wird in Zeile 11 vom *ClientContainer* ein *Client* angefragt. Der von *accept*() zurückgegebene Socket wird dann in Zeile 12 zur Initialisierung eines *HttpClients* genutzt. Letztlich wird der *HttpClient* noch an einen *ClientContainer* durch seine Methode *scheduleForExecution*() übergeben.

4.3.2 Der HttpClient

HttpClient ist eine Klasse, die das Interface *Client*, welches in der API festgelegt wurde, implementiert. Zu sehen ist die *HttpClient*-Klasse in Listing 45. Sie speichert den *Socket*, der zu dem Client gehört, den *InputStream* des *Sockets* und einen *BufferedOutputStream* der von dem *OutputStream* des Sockets erstellt wurde. Zusätzlich wird noch in dem *boolean keepAlive* festgehalten, ob die Verbindung nach einer kompletten Transaktion geöffnet bleiben soll. Der *HttpClient* muss die 2 Methoden *setSocket*() und *getInstance*() von *Client* implementieren.

```
1  public class HttpClient implements Client {
2
3      protected Socket s;
4      protected InputStream in;
5      protected BufferedOutputStream out;
6
7      private boolean keepAlive;
8
9      @Override
10     public void setSocket(Socket socket) {
11         s = socket;
12     }
13
14     @Override
15     public Client getInstance() {
16         return new HttpClient();
17     }
18 }
```

Listing 45: HttpClient Variablen und geerbte Methoden

Da das Interface *Client* selbst das Interface *Runnable* erweitert, muss *HttpClient* ebenfalls die *run*()-Methode implementieren. Zu sehen ist diese in Listing 46. Da alles, was mit Netzwerkkommunikation zu tun hat, *IOExceptions* werfen kann, wird der gesamte Inhalt der Methode mit einem *try-catch*-Block umschlossen. Dabei wird bei *IOExceptions*

nichts getan außer die *run*()-Methode zu beenden, da diese meist durch abgebrochene oder fehlerhafte Übertragungen geworfen wird. Zusätzlich wird in Zeile 18 noch separat jede weitere *Exception* abgefangen. Dies ist nur für Debugging wichtig, da bei manchen JVMs keine Stacktraces ausgeschrieben werden, wenn ein Hintergrund-Thread mit einer Exception endet und sich dadurch schwer sehen lässt, ob und wo Fehler ausgelöst wurden. Anfänglich wird in Zeile 4 die *init*()-Methode aufgerufen. In dieser wird als erstes der Timeout des Sockets festgelegt (Zeile 24).

```
1  @Override
2  public void run() {
3      try{
4          init();
5
6          do {
7              try {
8                handle();
9              }catch(SocketTimeoutException e) {
10                 keepAlive = false;
11             }
12         } while(keepAlive);
13
14         out.flush();
15         out.close();
16         s.close();
17     }catch(IOException e) {
18     }catch(Exception e) {
19         e.printStackTrace();
20     }
21 }
22
23 private void init() throws IOException {
24     s.setSoTimeout(60000);
25
26     in = s.getInputStream();
27     out = new BufferedOutputStream(s.getOutputStream());
28 }
```

Listing 46: HttpClient run() und init()

Dies ist dafür nötig, dass die Kommunikation nach dem Ablauf des festgelegten Timeouts eine *SocketTimeoutException* wirft. Diese macht es einfach, persistente HTTP-Verbindungen zu verwalten. Zusätzlich wird der *InputStream* des *Sockets* gespeichert und ein *BufferedOutputStream* aus dem *OutputStreams* des *Sockets* erstellt.

Danach wird (Zeile 6 bis 12) in einer Schleife die *handle*()-Methode des *HttpClients* aufgerufen. Allerdings nur so lange, wie *keepAlive* auf *true* gesetzt ist. Wenn eine *SocketTimeoutException* geworfen wird, wird *keepAlive* auf *false* gesetzt und die Schleife durchbrochen, wodurch der *OutputStream* und der *Socket* geschlossen werden.

Bevor die *handle*()-Methode behandelt wird, ist es erstmal nötig die *getRequest*()-Methode anzusehen. Diese versucht aus den Daten im *InputStream* ein *Request*-Objekt zu erstellen. Dafür wird die statische Methode *parseHeader*() von *HeaderParser*[1] aufgerufen (Zeile 4). Außerdem wird die IP-Adresse des Sockets im *Request*-Objekt festgelegt.

```
1  private Request getRequest() throws MalformedHeaderException, IOException {
2  try {
3      Request request = HeaderParser.parseHeader(in);
4      request.setRemoteAddress(s.getInetAddress().getHostAddress());
5
6      boolean keep = request.getRequestHeader().getConnection().equals("keep-alive");
7      boolean close = request.getRequestHeader().getConnection().equals("close");
8      boolean http11 = request.getVersion().equals("HTTP/1.1");
9
10     keepAlive = !close && (keep || http11);
11
12     return request;
13 } catch (MalformedHeaderException e) {
14     Response response = new Response(out, true);
15     response.setCode(e.getMessage().substring(e.getMessage().lastIndexOf(";") + 1));
16     response.setVersion("1.1");
17     response.getHeaders().setContentLength(0);
18
19     response.getHeaders().addConnection("close");
20
21     response.close();
22     e.printStackTrace();
23
24     keepAlive = false;
25
26     throw new MalformedHeaderException(e.getMessage(), e);
27 }
28 }
```

Listing 47: HttpClient getRequest()

In Zeile 6 wird geschaut, ob der Connection-Header auf keep-alive gesetzt ist und danach ob er auf close gesetzt ist. Nachfolgend wird überprüft, ob die Anfrage mit Version 1.1 gestellt wurde. Mit diesen drei Daten wird entschieden, ob *keepAlive* auf *true* gesetzt werden soll.

Die Formel in Zeile 10 kommt durch Vereinfachung einer Wahrheitstabelle mit boolescher Algebra zu Stande. Die volle Wahrheitstabelle ist in Tabelle 7 zu sehen.

[1] Erklärt in 4.3.3

Die Tabelle hat 3 Eingangsparameter:

- a ... Connection: keep-alive ist gesetzt,
- b ... Connection: close ist gesetzt und
- c ... HTTP-Version ist 1.1.

Und einen Ausgangsparameter:

- y ... keepAlive wird gesetzt.

a	b	c	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	-
1	1	1	-

Tabelle 7: Wahrheitstabelle keepAlive

Wenn die HTTP-Version 1.1 ist, ist keep-alive der Standard. Wenn das Header-Feld keep-alive gesetzt ist, soll die Verbindung geöffnet bleiben und wenn das Header-Feld close gesetzt ist, soll die Verbindung geschlossen werden. Da keep-alive und close nicht gleichzeitig gesetzt sein können, können die letzten 2 Zeilen ignoriert werden. Daraus lässt sich die folgende Formel schließen:

$$y = (\neg a \wedge \neg b \wedge c) \vee (a \wedge \neg b \wedge \neg c) \vee (a \wedge \neg b \wedge c)$$

Da b immer negiert ist lässt sich dieses ausklammern, wodurch diese Formel entsteht:

$$y = \neg b \wedge ((\neg a \wedge c) \vee (a \wedge \neg c) \vee (a \wedge c))$$

Und da der innere Teil der Klammern äquivalent zu $a \vee c$ ist, entsteht die endgültige Formel:

$$y = \neg b \wedge (a \vee c)$$

a	b	c	$\neg b$	$a \vee c$	y
0	0	0	1	0	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	1	0	1	0
1	0	0	1	1	1
1	0	1	1	1	1

Tabelle 8: Wahrheitstabelle Vergleich

Wie man in Tabelle 8 sehen kann, ist die Formel äquivalent zu der originalen Formel.

Nachdem die Werte festgelegt wurden, wird das Request-Objekt zurückgegeben. Im Falle, dass die Anfrage einen syntaktischen Fehler enthält, wird eine *MalformedHeaderException* geworfen. In dem *catch*-Block, der diesen verarbeitet, wird eine Fehlerantwort erstellt. Dabei wird in Zeile 15 der Fehlercode aus der Nachricht der Exception herausgefiltert. Danach wird die Version der Antwort auf 1.1 gesetzt, die Länge auf 0 festgelegt und gesagt, dass die Verbindung geschlossen werden wird. Folglich wird eine neue Exception mit der vorherigen als Auslösegrund geworfen.

```
1  private void handle() throws IOException {
2      Request request;
3      try {
4          request = getRequest();
5      } catch(MalformedHeaderException e) {
6          return;
7      }
8
9      try {
10         HttpHandler handler = (HttpHandler)HandlerContainer.getContainer().getHandler("defaultHttp");
11
12         Map<String, Object> options = new HashMap<>();
13         options.put("socketTimeout", s.getSoTimeout() / 1000);
14
15         handler.handleHttp(out, request, options);
16     } catch(NotImplementedException e) {
17         ...
18     } catch (Exception e) { //on any error
19         ...
20     }
21 }
```

Listing 48: HttpClient handle()

In Zeile 4 wird als erstes die eben behandelte *getRequest*()-Methode aufgerufen. Wenn diese fehlschlägt, wird entweder eine *IOException* oder eine *MalformedHeaderException* geworfen. Danach wird in Zeile 10 der *defaultHttp*-Handler von dem *HandlerContainer* abgerufen. Folgend wird eine *Map* erstellt, die weitere Information für den Handler beinhaltet, in diesem Fall nur den Timeout (Zeile 12 bis 13). Jetzt wird die *handleHttp*()-Methode des *Handlers* aufgerufen, der sich um das weitere Verarbeiten der HTTP-Anfrage kümmert.

4.3.3 Der HeaderParser

Der *HeaderParser* kümmert sich um das Parsen des Http-Headers. Dafür lädt er den Header Zeile für Zeile interpretiert diese. Er erstellt ein *Request*-Objekt, setzt alle Anfragedaten und fügt alle Header-Felder in das *RequestHeader*-Objekt, dass in *Request* erstellt wird, ein. Aufgerufen wird der Parser durch die statische Methode *parseHeader*(), die in Listing 49 zu sehen ist.

In Zeile 5 wird als erstes das *Request*-Objekt erstellt. Danach wird aus dem übergebenen *InputStream* ein *DataInputStream* erstellt.

Es wurde ein *DataInputStream* gewählt, da er sowohl Text-, als auch Binärdaten lesen kann. Es wird zwar davon abgeraten, die *readLine*()-Methode zu nutzen, da die Bytes nicht richtig in Chars konvertiert werden. [31] Es hat sich jedoch nach einigen Tests ergeben, dass dies nur bei Chars, die mehrere Bytes umfassen auftritt. Im HTTP-Standard ist allerdings festgelegt, dass Http-Nachrichten im US-ASCII-Format behandelt werden müssen [2]. Somit sind also alle Chars nur 1 Byte groß. Folglich kann also die *readLine*()-Methode ohne Probleme eingesetzt werden.

Nachdem der *Stream* erstellt wurde, wird die Anfrage-Zeile durch Aufruf der *getRequestLine*()-Methode ausgelesen. Diese wird in Zeile 9 durch Aufruf der *parseRequestLine*() interpretiert. Im Anschluss werden in *getHeaderFields*() alle Header-Felder gelesen und in das übergebene *RequestHeader*-Objekt eingetragen. Folgend wird noch der *DataInputStream* in das Request-Objekt durch *setInputStream*() eingefügt. Dieser wird zum Lesen des Nachrichtenkörpers benötigt. Zum Schluss wird noch das *Request*-Objekt zurückgegeben.

```
1  public class HeaderParser {
2
3      public static Request parseHeader(InputStream inputStream)
4        throws IOException, MalformedHeaderException {
5          Request request = new Request();
6          DataInputStream in = new DataInputStream(inputStream);
7          String requestLine = getRequestLine(in);
8
9          parseRequestLine(requestLine, request);
10         getHeaderFields(in, request.getRequestHeader());
11
12         request.setInputStream(in);
13
14         return request;
15     }
16 }
```

Listing 49: HeaderParser parseHeader()

Die Methode *getRequestLine*() ist in Listing 50 zu sehen. In Zeile 4-10 wird versucht eine Zeile zu lesen. Dies geschieht durch Aufrufen der *readLine*()-Methode. Diese gibt eine gelesene Zeile zurück oder *null,* wenn das Ende des Streams erreicht wurde. Bevor die gelesene Zeile weiter verwendet wird, wird als erstes überprüft, ob diese nicht leer ist, indem in Zeile 7 *isEmpty*() des Strings aufgerufen wird. Dies ist nötig, da bei offen gehaltenen Verbindungen noch Leerzeilen der vorherigen Nachricht vorhanden sein können. Falls nach 10 Versuchen die Anfrage-Zeile noch immer nicht gelesen werden

konnte, wird eine *MalformedHeaderException* geworfen und wenn der String *null* ist, wird eine IOException geworfen. Letztlich wird in Zeile 19 nur noch überflüssiger Whitespace entfernt und die Anfrage-Zeile zurückgegeben.

```
1  private static String getRequestLine(DataInputStream in) throws MalformedHeaderException, IOException {
2          String requestLine = "";
3
4          for(int i = 0; i < 10; i++) {
5              requestLine = in.readLine();
6
7              if(requestLine != null && !requestLine.isEmpty()) {
8                  break;
9              }
10         }
11
12         if(requestLine == null) {
13             throw new IOException();
14         }
15         if(requestLine.isEmpty()) {
16             throw new MalformedHeaderException("invalid Requestline;400");
17         }
18
19         return requestLine.trim();
20     }
```

Listing 50: HeaderParser - getRequestLine()

Die gelesene Anfrage-Zeile wird, nachdem sie gelesen wurde, in der *parseRequestLine*()-Methode, die in Listing 51 zu sehen ist, geparst. Dafür wird die Anfrage-Zeile und das *Request*-Objekt übergeben. Als erstes wird die Zeile aufgeteilt. Dafür wird die *split*()-Methode des Strings benutzt. Aufgeteilt wird hierbei nach Leerzeichen. Dabei sollten 3 Teile entstehen:

- die Methode,
- die URI und
- die Version.

Dies wird in Zeile 5 überprüft. Wenn es nicht 3 Teile sind, handelt es sich nicht um eine korrekte Anfrage-Zeile und es wird eine *MalformedHeaderException* geworfen.
Nachdem 3 Teile bestätigt wurden, wird als erstes in Zeile 9 die Methode aus den Teilen entnommen. Diese sollte der 1. Split sein. Der 2. Split sollte die URI enthalten, diese wird mit dem *URLDecoder* decodiert. Nun wird noch die Version, die in Split 3 sein sollte, entnommen.
Als nächstes wird die Methode und die Version in das *Request*-Objekt eingefügt. Dann wird in Zeile 16 überprüft, ob die URI ein „*?*“ beinhaltet. Ein „*?*“ in der URI bedeutet, dass eine Query in der URI vorhanden ist. Die Query befindet sich hinter dem „*?*“ und der Dateibezeichner vor diesem, weshalb in Zeile 17 und 18 die URI und die Query separat nach dem Splitten gesetzt werden. Wenn es keine Query gibt, wird die URI normal gesetzt.

```
1 private static void parseRequestLine(String requestLine, Request request)
2   throws MalformedHeaderException, UnsupportedEncodingException {
3     String[] requestSplits = requestLine.split(" ");
4
5     if (requestSplits.length != 3) {
6         throw new MalformedHeaderException("invalid Requestline;400");
7     }
8
9     String method = requestSplits[0];
10    String uri = URLDecoder.decode(requestSplits[1], "UTF-8");
11    String version = requestSplits[2];
12
13    request.setMethod(method);
14    request.setVersion(version);
15
16    if(uri.contains("?")) { //query exists
17        request.setUri(uri.split("\\?")[0]);
18        request.getQuery().addQuery(uri.split("\\?")[1]);
19    } else {
20        request.setUri(uri);
21    }
22 }
```

Listing 51: HeaderParser - parseRequestLine()

Nachdem die Anfrage-Zeile geparst wurde, müssen noch die Header-Felder folgen. Gelesen werden diese in der *getHeaderFields*()-Methode (Listing 52). Das Lesen der Header erfolgt in einer *while*-Schleife. In dieser wird als erstes in Zeile 4 die nächste Header-Zeile gelesen und der Whitespace entfernt. Danach wird geschaut, ob die Zeile leer ist. Ist sie leer, bedeutet das, dass der Header abgeschlossen ist, andernfalls wird das Header-Feld in Zeile 9 in zwei Teile aufgeteilt, in den Namen und den Wert. Der Feld-Name befindet sich immer im ersten Teil, im zweiten Teil befindet sich immer der Wert.

```
1 private static void getHeaderFields(DataInputStream in, RequestHeader rh)
2   throws IOException, MalformedHeaderException {
3     while (true) {
4         String line = in.readLine().trim();
5
6         if (line.length() == 0)
7             break;
8
9         String[] splits = line.split(":", 2);
10
11        String fieldName = splits[0];
12        String fieldValue;
13        if (splits.length == 2) {
14            fieldValue = splits[1];
15
16            fieldValue = fieldValue.trim();
17        } else if (splits.length == 1)
18            fieldValue = "";
19        else {
20            throw new MalformedHeaderException("invalid header field;400");
21        }
22
23        insertHeaders(rh, fieldName, fieldValue);
24    }
25 }
```

Listing 52: HeaderParser - getHeaderFields()

In Zeile 13 wird geschaut, ob es zwei Teile sind und wenn es zwei sind, wird der Wert auf den Inhalt des zweiten Feldes gesetzt. Wenn es nur ein Feld gibt, wird der Inhalt auf einen leeren String gesetzt. Andernfalls wird eine *MalformedHeaderException* geworfen. Nachdem Name und Wert festgelegt wurden, wird die Methode *insertHeaders*() (Listing 53) aufgerufen. Diese kümmert sich um das Einfügen der Header-Felder. Dies wird so lange wiederholt, wie es Header-Felder gibt.

```
1 private static void insertHeaders(RequestHeader rh, String name, String value)
2   throws MalformedHeaderException {
3     if(name.equals("Warning")) {
4         rh.addWarning(new HttpWarning(value));
5     } else {
6         rh.addHeader(name, value);
7     }
8 }
```

Listing 53: HeaderParser - insertHeaders()

In dieser Methode wird unterschieden, ob es sich um ein *Warning*-Header-Feld handelt, oder um ein anderes Feld. Diese Unterscheidung ist nötig, da das *Warning*-Feld kein einzelnes kommasepariertes Feld ist, es stattdessen mehrere *Warning*-Felder geben kann. Die anderen Felder haben jedoch entweder kommaseparierte oder einzelne Werte. Hinzugefügt werden Header durch Aufruf der *addHeader*()-Methode, des *RequestHeader*-Objekts. Diese Methode nimmt den Header-Namen und den Header-Wert als Parameter an.

4.3.4 Der DefaultHttpHandler

Die Klasse *DefaultHttpHandler* (Listing 54) erweitert die Klasse *HttpHandler*[1]. Der Handler unterstützt die Methoden GET, HEAD und POST. Dafür werden die Methoden *doGET*() und *doPOST*() überschrieben. Die *doHEAD*()-Methode muss nicht überschrieben werden, da diese standardmäßig *doGET*() aufruft. Außerdem werden die Methoden *getHandlerName*(), *getInstance*() und *handleOptions*() überschrieben.

```
1  public class DefaultHttpHandler extends HttpHandler {
2
3      public void doGET(Response resp, Request request) throws IOException {...}
4      public void doPOST(Response resp, Request request) throws IOException {...}
5
6      public String getHandlerName() {
7          return "defaultHttp";
8      }
9
10     public DefaultHttpHandler getInstance() {
11         return new DefaultHttpHandler();
12     }
13
14     protected void handleOptions(Request request, Response response,
15                                  Map<String, Object> options) {
16         int to = (int) options.getOrDefault("socketTimeout", 0);
17
18         response.getHeaders().setHeader("Connection", "keep-alive");
19         response.getHeaders().setHeader("Keep-Alive", "timeout=" + to);
20     }
21 }
```

Listing 54: DefaultHttpHandler - Überschriebene Methoden

Als Name des Handlers wurde *defaultHttp* gewählt. Dieser wird von der Methode *getHandlerName*(), die in Zeile 6 zu sehen ist, zurückgegeben. Die Methode *getInstance*() gibt eine durch *new* erstellte Instanz dieses Handlers zurück.

Die *handleOptions*()-Methode, die vor den *do*()-Methoden aufgerufen wird, nimmt weitere Einstellungen am *Response*-Objekt vor. Als erstes wird von der übergebenen *Map* der Timeout abgefragt (Zeile 16). Danach wird der Header *Connection* auf *keep-alive* gesetzt und der Header *Keep-Alive* mit dem übergebenen Timeout versehen.

[1] Siehe Abschnitt 4.1.2

```
1  public void doGET(Response resp, Request request) throws IOException {
2      resp.setVersion("1.1");
3      resp.setCode("200");
4
5      resp.getHeaders().setHeader("Accept-Ranges", "none");
6
7      try {
8          handle(resp, request);
9      } catch (NoHandlerException e) {
10         resp.setCode("500");
11         e.printStackTrace();
12     }
13 }
14
15 public void doPOST(Response resp, Request request) throws IOException {
16     doGET(resp, request);
17 }
```

Listing 55: DefaultHttpHandler - doGet() und doPost()

In Listing 55 werden die *doGet*()- und *doPost*()-Methoden beschrieben. In *doPost*() wird lediglich *doGet*() aufgerufen. Dies ist notwendig, da dieser Handler keine Webseiten dynamisch generiert, ein später aufgerufener *FileHandler* jedoch CGI[1]-Skripte oder Servlets zum Generieren von Webseiten nutzen könnte.

Die *doGET*()-Methode kümmert sich um das Setzen der genutzten HTTP-Version 1.1 und um das Setzen des Antwortcodes 200 (Zeilen 2 und 3). Danach wird der *Accept-Ranges*-Header auf *none* gesetzt. Falls ein *FileHandler* Range-Anfragen unterstützen will, kann er den Inhalt des Headers später auf einen anderen angemessenen Wert setzen. Danach wird *handle*() aufgerufen. Wenn in dieser Methode etwas schief läuft wird der Antwortcode auf 500 gesetzt.

```
1  private void handle(Response resp, Request request) throws NoHandlerException, IOException {
2      @SuppressWarnings("unchecked")
3      ArrayList<String> fileHandlers = (ArrayList<String>)
4        Options.getOption("fileHandling", new DefaultModule());
5
6      for(String fileHandler : fileHandlers) {
7          FileHandler handler = (FileHandler)HandlerContainer.getContainer().getHandler(fileHandler);
8
9          if(handler.handle(request, resp) == FinishState.FINISHED) {
10             break;
11         }
12     }
13 }
```

Listing 56: DefaultHttpHandler - handle()

In dieser Methode werden als erstes die in der Konfigurationsdatei festgelegten Handler als Liste abgefragt. Dafür wird in Listing 56, Zeile 4 *getOption*() von der Klasse *Options* aufgerufen.

[1] Common Gateway Interface

Dann wird von Zeile 6-12 über alle festgelegten *FileHandler* iteriert. Diese werden in Zeile 7 vom *HandlerContainer* abgerufen. Danach wird die handle()-Methode von diesen aufgerufen und wenn diese den Rückgabewert *FINISHED* zurückgibt, wird das Iterieren beendet.

4.3.5 Der DefaultFileHandler

Der *DefaultFileHandler* (Listing 57) ist der *FileHandler,* der standardmäßig mit dem Server ausgeliefert wird. Er unterstützt das Übertragen von Dateien, Range-Anfragen, erstellt eine simple Übersicht von Ordnern und unterstützt das Cachen von Dateien, um IO-Operationen zu minimieren.

```
1  public class DefaultFileHandler extends FileHandler {
2
3      protected FinishState handleFile(Request request, Response response, File file) {...}
4      protected FinishState handleRange(Request request, Response response, File file) {...}
5      protected FinishState handleDirectory(Request request, Response response, File file) {...}
6
7      public DefaultFileHandler getInstance() {
8          return new DefaultFileHandler();
9      }
10
11     public String getHandlerName() {
12         return "defaultFile";
13     }
14 }
```

Listing 57: DefaultFileHandler - Überschriebene Methoden

Als Name des Handlers wurde *defaultFile* gewählt. Dieser wird in Zeile 12 beim Aufruf der *getHandlerName*()-Methode zurückgegeben. Zusätzlich wird noch bei Aufruf der *getInstance*()-Methode eine neue Instanz des Handlers zurückgegeben. (Zeile 7-8)

```
1  protected FinishState handleDirectory(Request request, Response response, File file)
2    throws IOException {
3      if (file.isDirectory() && !request.getUri().endsWith("/")) {
4          response.setCode("301");
5          response.getHeaders().setHeader("Location", request.getUri() + "/");
6          response.getHeaders().setHeader("Content-Length", 0);
7
8          return FinishState.FINISHED;
9      }
10
11     ...
12 }
```

Listing 58: DefaultFileHandler - handleDirectory() Weiterleitung

Die *handleDirectory*()-Methode kümmert sich um das dynamische Erstellen von Webseiten, die den Inhalt eines Ordners und weiterführende Links anzeigen. Dafür ist allerdings nötig, dass die Anfrage von der richtigen URI ausgeht, da es sonst zu falschen Weiterleitungen kommen kann. Beispielsweise würden für den Server die URIs

„/directory“ und „/directory/“ beide auf denselben Ordner verweisen. Allerdings werden die URIs in einem Browser bei Weiterleitungen unterschiedlich behandelt, obwohl es quasi die gleiche Ausgangswebseite ist. Wie diese behandelt werden ist ein Tabelle 9 zu sehen.

Basis-URI	Weiterleitung	Ziel-URI
/directory	test.html	/test.html
/directory/	test.html	/directory/test.html

Tabelle 9: Weiterleitungen

Damit Weiterleitungen richtig funktionieren, ist es also nötig, den Browser an die korrekte URI weiterzuleiten. Wie dies funktioniert ist in Listing 58 zu sehen. Dabei wird als erstes überprüft, ob die URI mit einem „/“ endet. Wenn nicht, ist eine Weiterleitung nötig. Dafür wird in Zeile 4 der Antwortcode auf 301 (Moved Permanently) gesetzt und in Zeile 5 die Ziel-URI in das *Location*-Header-Feld eingetragen. Danach wird nur noch Content-Length auf 0 gesetzt, um zu zeigen, dass es keinen Nachrichtenkörper gibt und der Wert *FINISHED* zurückgegeben. Der Browser leitet danach weiter und stellt eine Anfrage mit der korrekten Basis-URI.

```
1 protected FinishState handleDirectory(Request request, Response response, File file)
2   throws IOException {
3     ...
4     response.getHeaders().setHeader("Accept-Ranges", "none");
5     response.getHeaders().setHeader("Content-Type", "text/html; charset=utf-8");
6
7     ResponseOutputStream out = response.getOutputStream();
8
9     out.write("<!doctype html>\n<http>");
10    out.write("<h1>" + file.getName() + "</h1>");
11
12    out.write(makeFileList(file));
13
14    out.write("<br>");
15    out.write("<hr>");
16
17    out.write("<img src=\" ");
18    out.write(StaticFiles.getFile("logo.png"));
19    out.write("\" />");
20
21    out.write(ResponseHeader.getServer());
22    out.write("</html>");
23
24    return FinishState.FINISHED;
25 }
```

Listing 59: DefaultFileHandler - handleDirectory() Generierung

Nachdem bestätigt wurde, dass die Basis-URI korrekt ist, kann die Seite generiert werden. Dies wird in Listing 59 dargestellt. Als erstes wird in den Zeilen 4-5 die korrekten Header-Felder festgelegt. Danach wird in Zeile 7 der *OutputStream* der Response angefragt. Folgend wird nach und nach der HTML-Code erstellt und in den *OutputStream* geschrieben. Dafür ist es nur nötig, diesen in der *write*()-Methode auszuschreiben. Die

makeFileList()-Methode, die in Zeile 12 aufgerufen wird, kümmert sich um das Erstellen eines Strings, der den nötigen HTML-Code mit allen Weiterleitungen beinhaltet. In Zeile 18 wird von der Klasse *StaticFiles* ein Logo mit Base64-encoding abgefragt.
Nachdem alles ausgeschrieben wurde, wird der Status *FINISHED* zurückgegeben. Eine so generierte Webseite ist in Abbildung 11 zu sehen.

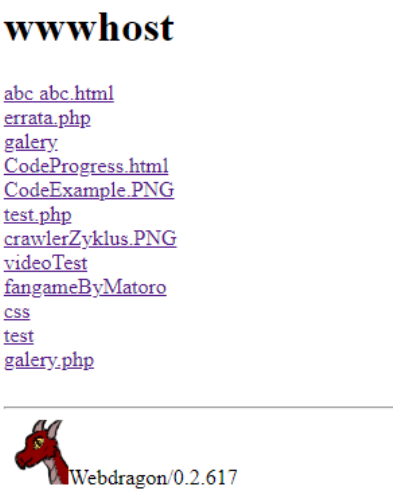

Abbildung 11: DefaultFileHandler - Generierte Webseite

Die *handleFile*()-Methode (Listing 60) kümmert sich um das Übertagen von Dateien. Dabei wird unterschieden, ob diese Datei gecached werden kann oder nicht. (Zeile 9)

```
1  protected FinishState handleFile(Request request, Response response, File file)
2    throws IOException {
3      Resource resource = ResourceHolder.getResource(request, file);
4
5      response.getHeaders().setHeader("Accept-Ranges", "bytes");
6      response.getHeaders().setHeader("Last-Modified", resource.getLastModified());
7      response.getHeaders().setHeader("Content-Type", resource.getMime());
8
9      if (file.length() < CacheHolder.getCacheableSize()) {   //cached
10         ...
11     } else {    //uncached
12         ...
13     }
14
15     return FinishState.FINISHED;
16 }
```

Listing 60: DefaultFileHandler - handleFile() Übersicht

Dafür wird als erstes in Zeile 3 ein Resource-Objekt von der Klasse *ResourceHolder* angefragt. Danach werden die Header-Felder Accept-Ranges, Last-Modified und Content-Type gesetzt. Die Werte für Last-Modified und Content-Type werden in *ResourceHolder* initialisiert. Danach wird in Zeile 9 überprüft, ob die Dateilänge kleiner oder größer als die cachebare Länge[1] ist. Danach wird jeweils unterschiedlich verfahren. Nachdem die Datei übertragen wurde, wird *FINISHED* zurückgegeben.

[1] Die maximale Länge einer cachebaren Datei beträgt 1MB.

```
1 if (file.length() < CacheHolder.getCacheableSize()) {   //cached
2     ResponseOutputStream out = response.getOutputStream();
3
4     byte[] content = CacheHolder.getContent(file);
5     response.getHeaders().setHeader("Content-Length", content.length);
6     out.write(content);
7 } else {
8     ...
9 }
```

Listing 61: DefaultFileHandler - handleFile() cachebare Datei

In Listing 61 sieht man das Verfahren bei cachebaren Dateien. Als erstes wird in Zeile 2 der *OutputStream* der *Response* angefragt. Danach wird von der *CacheHolder*-Klasse durch Aufruf ihrer *getContent*()-Methode der gesamte Dateiinhalt als Byte-Array zurückgegeben. Im Anschluss wird noch das Content-Length-Feld gesetzt und das gesamte Byte-Array ausgeschrieben.

```
1  public class CacheHolder {
2
3      private final static Cache<String, byte[]> cache = new SimpleCache<>();
4      private final static int cacheableSize = 1024*1024;
5
6      protected static int getCacheableSize() {return cacheableSize;}
7
8      protected static byte[] getContent(File f)
9        throws FileNotFoundException, IOException {
10         byte[] content = cache.get(f.getAbsolutePath());
11
12         //if not in cache add it
13         if(content == null) {
14             content = Files.readAllBytes(f.toPath());
15             cache.put(f.getAbsolutePath(), content);
16         }
17
18         return content;
19     }
20 }
```

Listing 62: CacheHolder

Die Klasse *CacheHolder*, die in Listing 62 zu sehen ist, verwaltet ein *Cache*-Objekt, indem sie sich um dessen Befüllen kümmert. Dies geschieht, indem bei Aufruf der *getContent*()-Methode der Inhalt des *Caches* als erstes abgefragt wird (Zeile 10). Der absolute Pfad der Datei wird dabei als Schlüssel verwendet. Wenn der *Cache* ein Byte-Array zurückgibt, wird dieses einfach zurückgegeben. Wenn er allerdings null zurückgibt bedeutet das entweder, dass der Inhalt nie hinzugefügt wurde, oder dass er vom Garbage-Collector entfernt wurde. In beiden Fällen wird in Zeile 14 durch Aufruf von *Files.readAllBytes*() der gesamte Dateiinhalt gelesen. Dieser wird danach im Cache gelagert und zurückgegeben.

```
1  if (file.length() < CacheHolder.getCacheableSize()) {
2      ...
3  } else {    //uncached
4      try (FileInputStream fis = new FileInputStream(file)) {
5          ResponseOutputStream out = response.getOutputStream();
6          response.getHeaders().setHeader("Content-Length", file.length());
7
8          byte[] b = new byte[1024*1024];
9          int read;
10
11         while ((read = fis.read(b)) != -1) {
12             out.write(b, 0, read);
13         }
14     }
15 }
```

Listing 63: DefaultFileHandler - handleFile() nicht cachebare Datei

Eine Datei, die zu groß für den Cache ist, wird immer direkt vom Filesystem eingelesen. Wie dies funktioniert sieht man in Listing 63. Dafür wird als erstes in Zeile 4 ein *FileInputStream* erstellt, der von der Datei lesen kann. Danach wird der OutputStream von Response abgefragt und der Content-Length-Header gesetzt. Als nächstes wird ein Byte-Array als Buffer angelegt (Zeile 8), in dem die Daten gelesen werden. In Zeile 11 wird begonnen die Datei einzulesen. Dafür wird die *read*()-Methode des *FileInputStreams* benutzt. Diese füllt die gelesenen Bytes in den Buffer und gibt die Anzahl der gelesenen Bytes zurück. Wenn die Anzahl minus eins ist, heißt das, dass das Ende der Datei erreicht wurde. Nach jedem Lesedurchlauf werden die gelesenen Bytes in den *OutputStream* geschrieben.

Die *handleRange*()-Methode (Listing 64) kümmert sich um das Antworten auf Range-Anfragen.

```
1  protected FinishState handleRange(Request request, Response response, File file) throws IOException {
2      Resource resource = ResourceHolder.getResource(request, file);
3
4      response.getHeaders().setHeader("Last-Modified", resource.getLastModified());
5      response.getHeaders().setHeader("Content-Type", resource.getMime());
6
7      DefaultFileRangeHandler.handleRanges(request, response, file);
8
9      return FinishState.FINISHED;
10 }
```

Listing 64: DefaultFileHandler - handleRange()

Sie holt, wie auch die *handleFile*()-Methode, als erstes ein Resource-Objekt von der *ResourceHolder*-Klasse. Danach werden die Header Last-Modified und Content-Type festgelegt. Das richtige Abarbeiten der Range-Anfrage wird in der Klasse

DefaultFileRangeHandler übernommen. deren *handleRanges*()-Methode wird in Zeile 7 aufgerufen. Nachdem diese ausgeführt wurde, wird der Staus FINISHED zurückgegeben. Das Bearbeiten der Range-Anfragen wurde in eine andere Klasse verschoben, da dies die Lesbarkeit des Codes verbessert. Die Methode *handleRanges*() der *DefaultFileRangeHandler*-Klasse ist in Listing 65 zu sehen.

```
1 protected static void handleRanges(Request request, Response response, File f)
2   throws IOException {
3     List<Range> ranges = request.getRequestHeader().getRanges(f.length());
4
5     if(ranges.size() == 1) {
6         handleSingleRange(ranges.get(0), response, f);
7     } else {
8         handleMultiRange(ranges, response, f);
9     }
10 }
```

Listing 65: DefaultFileRangeHandler - handleRanges()

In dieser werden alle *Ranges,* die in der Anfrage festgelegt wurden, in Zeile 3 als Liste abgerufen. Danach wird unterschieden, ob es sich um eine *Range* oder um mehrere handelt. Je nachdem wird entweder die *handleSingleRange*() oder die *handleMultiRange*()-Methode aufgerufen.

```
1 private static void handleSingleRange(Range range, Response response, File f)
2   throws IOException {
3     ResponseOutputStream out = response.getOutputStream();
4
5     if(range.getBeginning() > range.getEnd() ||
6        range.getBeginning() > range.getResourceSize()) { //invalid
7         response.setCode("416");
8         response.getHeaders().setContentRangeLength(range.getResourceSize());
9
10        return;
11    }
12
13    response.setCode("206");
14    response.getHeaders().setHeader("Content-Range", range);
15    response.getHeaders().setHeader("Content-Length", range.getLength() + "");
16
17    writeRange(range, f, out);
18 }
```

Listing 66: DefaultFileRangeHandler - handleSingleRange()

Die *handleSingleRange*()-Methode (Listing 66) überprüft als erstes in den Zeilen 5 - 6, ob es sich um eine valide Range-Anfrage handelt. Wenn der Startwert der Anfrage höher als der Endwert ist oder der Startwert größer als die Ressourcengröße ist, handelt es sich um eine fehlerhafte Anfrage. In diesem Fall wird der Antwortcode auf 416 (Range Not

Satisfiable) gesetzt und in Zeile 8 die Ressourcenlänge festgelegt. Im Anschluss kehrt die Methode zurück.

Wenn die Anfrage valide ist, wird der Code 206 (Partial Content) und die Header-Felder Content-Range und Content-Length festgelegt (Zeilen 13 - 15). Im Anschluss wird die *writeRange*()-Methode aufgerufen, die nach der *handleMultiRange*()-Methode erklärt wird.

Die *handleMultiRange*()-Methode (Listing 67) antwortet auf eine Range-Anfrage mit mehreren Ranges mit dem Content-Type multipart/byteranges[1]. Dafür wird der Content-Type der Ressource und ein Separator benötigt. Diese werden in Zeile 5 - 6 festgelegt. Danach wird der Code auf 206 und der Content-Type der Nachricht gesetzt.

Es wird dann über jede *Range* iteriert und jeweils der Body für diese festgelegt (Zeilen 11 - 21). Auch hier wird in Zeile 19 die *writeRange*()-Methode genutzt.

```
1  private static void handleMultiRange(List<Range> ranges, Response response, File f)
2    throws IOException {
3      ResponseOutputStream out = response.getOutputStream();
4
5      MimeType contentType = response.getHeaders().getContentType();
6      String separator = "asdaefthtjkizkmafuoehluairopghaogroigaeroig";
7
8      response.setCode("206");
9      response.getHeaders().setHeader("Content-Type", "multipart/byteranges; boundary=" + separator);
10
11     for(Range r : ranges) {
12         if(r.getEnd() > r.getResourceSize())
13             r.setEnd(r.getResourceSize() - 1);
14
15         out.write("--" + separator + "\r\n");
16         out.write("Content-Type: " + contentType + "\r\n");
17         out.write("Content-Range: " + r.getUnit() + " " + r + "\r\n");
18         out.write("\r\n");
19         writeRange(r, f, out);
20         out.write("\r\n");
21     }
22
23     out.write("--" + separator + "--\r\n");
24 }
```

Listing 67: DefaultFileRangeHandler - handleMultiRange()

Die *writeRange*()-Methode ist in Listing 68 zu sehen. Diese kümmert sich um das Auslesen der angefragten Dateiabschnitte und um das Ausschreiben der Ranges. Auch hierfür wird der Cache verwendet, wenn die Dateien die passende Größe haben. In Zeile 2 wird dafür *getContent*() von *CacheHolder* aufgerufen. Da diese Methode *null* zurückgibt, wenn die Datei zu groß für den Cache ist, kann dadurch unterschieden werden, ob die Datei vom Filesystem gelesen werden muss oder nicht (Zeile 4). In Zeile

[1] Der Aufbau dieses Content-Types ist auf Seite 28 beschrieben

5 wird der Dateiinhalt, aus dem Cache ausgeschrieben. Dafür wird der Startwert der *Range* und die Länge der *Range* genutzt.

```
1  private static void writeRange(Range range, File f, ResponseOutputStream out) throws IOException {
2      byte[] content = CacheHolder.getContent(f);
3
4      if(content != null) {
5          out.write(content, (int)range.getBeginning(), (int)range.getLength());
6      } else {
7          try (FileInputStream fis = new FileInputStream(f)) {
8
9              long skipAmount = range.getBeginning();
10             long skipped = 0;
11
12             while(skipped < skipAmount) {
13                 skipped += fis.skip(skipAmount - skipped);
14             }
15
16             byte[] b = new byte[1024*1024];
17             int read;
18             int readSum = 0;
19
20             while (readSum < range.getLength()) {
21                     read = fis.read(b, 0, Math.min((int)range.getLength() - readSum, b.length));
22                     if(read == -1) {
23                         return;
24                     }
25
26                     out.write(b, 0, read);
27                     readSum += read;
28                 }
29         }
30     }
31 }
```

Listing 68: DefaultFileRangeHandler - writeRange()

In den Zeilen 7 - 25 wird ein *FileInputStream* geöffnet, der aus der betroffenen Datei lesen kann. Zu Beginn muss bis zum Startwert der *Range* alles übersprungen werden. Dafür wird die *skip*()-Methode des *FileInputStream* genutzt. Zu diesem Zweck wird in den Zeilen 9 - 10 in der Variable *skipAmount* gespeichert wie viele Byte insgesamt übersprungen werden müssen und in der Variable *skipped,* wie viele übersprungen wurden. Anschließend wird in einer Schleife (Zeile 12 - 14) wiederholt *skip*() aufgerufen und der Rückgabewert auf *skipped* addiert, bis *skipped* gleich *skipAmount* ist.
Danach werden die Variablen

- *b*, ein Byte-Array zum Puffern der gelesenen Bytes,
- *read*, ein Integer, der speichert, wie viele Bytes in einem Durchlauf gelesen wurden und
- *readSum*, ein Integer, der speichert, wie viele Bytes insgesamt gelesen wurden,

erstellt.
Jetzt wird in der *while*-Schleife (Zeile 20 - 28) so lange der Dateiinhalt gelesen, bis die Anzahl der gelesenen Bytes gleich der Länge der *Range* ist. In Zeile 21 wird vom

FileInputStream gelesen. Die Anzahl der gelesenen Bytes soll dabei entweder die Länge des Buffers oder die Restlänge betragen. Wenn in Zeile 22 festgestellt wurde, dass *read*() -1 zurückgab wird die Schleife durchbrochen. Der in den Buffer gelesene Inhalt, wird in Zeile 26 ausgeschrieben. Zum Schluss wird noch *readSum* erhöht. Sobald dieser Vorgang abgeschlossen ist, kehrt *writeRange*() zurück.

4.4 Das php-Modul

Das php-Modul ist dafür entwickelt worden, um zu testen, ob die API richtig mit dem Server arbeitet und der Server Module richtig finden und einbinden kann. Es ist ein plattformabhängiges Modul, dass auf Linuxgeräten funktioniert, die php-cgi installiert haben. Das Modul besitzt nur einen *Handler,* die Klasse *PhpHandler*. Dieser *Handler* ist ein *FileHandler*.

Dieser nutzt das CGI-Protokoll[1], um mit dem Programm php-cgi zu kommunizieren. Dieses Programm kümmert sich dann um das Verarbeiten einer Php-Datei und gibt das Ergebnis an den Server zurück. Der Server wertet es aus und sendet dieses dann an den Client.

In Listing 69 ist ein ganz einfaches Php-Beispiel gezeigt, das "phptest" und die unformatierte jetzige Serverzeit ausschreibt.

```
1 phptest
2
3 <?php
4     echo time();
5 ?>
```

Listing 69: test.php

Ein Aufruf der Seite test.php zeigt nun, dass das Ausführen des Skripts erfolgreich war. (Abbildung 12)

Abbildung 12: Aufruf test.php

[1] Mehr Informationen zu CGI: https://tools.ietf.org/html/rfc3875

5 Fazit

Im Verlauf dieser Arbeit wurde ein modularer HTTP-Webserver entwickelt, der den Namen „Webdragon“ erhalten hat. Dieser bietet die grundlegende Funktionen eines Webservers mit einem integrierten Caching-Verfahren. Weitere Funktionen können durch eine einfache API hinzugefügt werden.

Auch wenn der Server voll einsatzfähig ist, ist damit seine Entwicklung noch nicht abgeschlossen. Diese wird sich aber hauptsächlich auf die Weiterentwicklung von Modulen konzentrieren. Beispielsweise sollte das Standard-Modul noch so erweitert werden, dass es auf Conditionals reagieren kann, um die Serverlast zu verringern. Dafür wäre es nötig, dieses Modul mit Verfahren zu erweitern, die schnell ETags bilden und auswerten können.

Weiterhin sollte das php-Modul so verändert werden, dass es nicht mehr plattformabhängig ist und mit schnelleren, skalierbareren Protokollen, die mit dem php-cgi-Programm kommuniziert. Wichtig wäre zusätzlich noch die Unterstützung von HTTPS, für sichere Übertragung aller Daten und der HTTP-Version 2.0.

Wie man sehen kann steckt also noch viel Arbeit in der Weiterentwicklung des Servers und dessen Modulen.

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Glossar

User Agent

Als User Agenten bezeichnet man ein Client-Programm mit welchem ein Netzwerkdienst genutzt werden kann.

Webbrowser

Ein Webbrowser ist ein Computerprogramm zur Anzeige von Webseiten.

Crawler

Ein Crawler ist ein Programm, dass automatisch Webseiten durchsucht und diese analysiert.

Filesystem

Das Filesystem, auch Dateisystem genannt, ist die Schnittstelle zu Datenträgern. Über diese ist es möglich Dateien zu lesen, schreiben und zu löschen.

Cache

Ein Cache ist ein schneller Zwischenspeicher, der dazu eingesetzt wird langsame Zugriffe oder Berechnungen zu vermeiden.

Reflection

Reflection dient dazu, die Bestandteile von Klassen dynamisch zur Laufzeit zu analysieren und zu benutzen. [32]

Abkürzungsverzeichnis

HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IANA	Internet Assigned Numbers Authority
TCP	Transmission Control Protocol
URI	Uniform Resource Identifier
MIME	Multipurpose Internet Mail Extensions
GMT	Greenwich Mean Time
CRLF	Carriage Return Line Feed (Zeilenumbruch)
JVM	Java Virtual Machine
API	Application Programming Interface
OOM	Out of memory
JSON	JavaScript Object Notation
XML	Extensible Markup Language
CGI	Common Gateway Interface
IO	input/output

Quellenverzeichnis

[1] https://w3techs.com/technologies/overview/web_server/all Stand 03.03.2019

[2] https://tools.ietf.org/html/rfc7230 Stand 03.03.2019

[3] https://www.tutorialspoint.com/http/http_header_fields.htm Stand 03.03.2019

[4] https://tools.ietf.org/html/rfc3986#section-3 Stand 04.03.2019

[5] https://www.w3.org/Protocols/HTTP/AsImplemented.html Stand 05.03.2019

[6] https://tools.ietf.org/html/rfc1945 Stand 04.03.2019

[7] https://tools.ietf.org/html/rfc7540 Stand 04.03.2019

[8] https://tools.ietf.org/html/rfc7231 Stand 05.03.2019

[9] https://www.iana.org/assignments/message-headers/message-headers.xhtml Stand 16.03.2019

[10] https://tools.ietf.org/html/rfc6265 Stand 19.03.2019

[11] https://tools.ietf.org/html/rfc7234#section-5.2 Stand 20.03.2019

[12] https://www.w3.org/TR/html52/sec-forms.html#form-control-infrastructure-form-submission Stand 22.03.2019

[13] https://www.tutorialspoint.com/http/http_responses.htm Stand 04.04.2019

[14] https://tools.ietf.org/html/rfc7231#section-6 Stand 04.04.2019

[15] https://www.it-agile.de/wissen/agiles-engineering/unit-tests/ Stand 05.04.2019

[16] http://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/is-management/Systementwicklung/Hauptaktivitaten-der-Systementwicklung/Software-Implementierung/Testen-von-Software/Systemtest/index.html Stand 05.04.2019

[17] https://community.oracle.com/blogs/enicholas/2006/05/04/understanding-weak-references Stand 05.04.2019

[18] https://stackoverflow.com/a/4553642 Stand 05.04.2019

[19] https://tools.ietf.org/html/rfc850#section-2.1.4 Stand 20.04.2019

[20] https://stackoverflow.com/a/435568 Stand 09.05.2019

[21] https://tools.ietf.org/html/rfc7233#section-1 Stand 23.05.2019

[22] https://tools.ietf.org/html/rfc7233#section-3 Stand 24.05.2019

[23] https://tools.ietf.org/html/rfc7230#section-3.3 Stand 24.05.2019

[24] https://tools.ietf.org/html/rfc7233#section-4 Stand 24.05.2019

[25] https://tools.ietf.org/html/rfc7233#section-2.3 Stand 24.05.2019

[26] https://tools.ietf.org/html/rfc7231#section-7.1 Stand 24.05.2019

[27] https://tools.ietf.org/html/rfc7231#section-7.2 Stand 25.05.2019

[28] https://tools.ietf.org/html/rfc7232#section-2.3 Stand 25.05.2019

[29] https://tools.ietf.org/html/rfc7231#section-7.4 Stand 25.05.2019

[30] https://github.com/fangyidong/json-simple Stand 14.06.2019

[31] https://docs.oracle.com/javase/7/docs/api/java/io/DataInputStream.html#readLine() Stand 02.08.2019

[32] https://www.dpunkt.de/java/Die_Sprache_Java/Objektorientierte_Programmierung_mit_Java/70.html Stand 26.09.2019

[33] https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml Stand 17.10.2019
[34] https://www.elektronik-kompendium.de/sites/kom/0301201.htm Stand 17.10.2019
[35] https://tools.ietf.org/html/rfc2616 Stand 17.10.2019
[36] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control Stand 17.10.2019
[37] https://www.tutorialspoint.com/http/http_requests.htm Stand 17.10.2019
[38] https://tools.ietf.org/html/rfc7235 Stand 17.10.2019

Anhang A

Anlagen

CD-ROM mit folgendem Inhalt:

- Quellcode des Servers und der API
- Bachelorarbeit im PDF-Format

Anhang B

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegeben Literatur und Hilfsmitteln angefertigt habe. Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken als solche kenntlich gemacht habe.

Ort, Datum eigenhändige Unterschrift