AlexWebserver – Dokumentation

Dokumentation zur Einsendeaufgabe B für das Modul

„VII – Grundlagen Verteilte Informationsverarbeitung“

Einsendeaufgabencode:

B-VIINK01-XX1-N01

Student:

Alexander Nitter, Matrikelnummer 893372

Ort, Datum:

Kassel, 21.03.2016

Inhaltsverzeichnis

[1 Einführung 1](#_Toc446399194)

[1.1 Aufgabenstellung 1](#_Toc446399195)

[1.2 Rahmenbedingungen 1](#_Toc446399196)

[1.3 Projektbeschreibung 2](#_Toc446399197)

[2 Anforderungsdefinition 3](#_Toc446399198)

[2.1 Anforderungen Server 3](#_Toc446399199)

[2.2 Anforderungen Client 6](#_Toc446399200)

[3 Konzeption 7](#_Toc446399201)

[3.1 Einleitung 7](#_Toc446399202)

[3.2 Architekturübersicht 7](#_Toc446399203)

[3.3 Klassendiagramm der Shared-Bibliothek 8](#_Toc446399204)

[3.4 Klassendiagramm Server 9](#_Toc446399205)

[3.5 Tests 10](#_Toc446399206)

[4 Anhang I](#_Toc446399207)

[4.1 Aufgabenstellung I](#_Toc446399208)

[4.2 Quellenangaben III](#_Toc446399209)

Abbildungsverzeichnis

[Abbildung 1: schematischer Ablauf AlexWebserver 5](#_Toc446399271)

[Abbildung 2: Paketdiagramm AlexWebserver 7](#_Toc446399272)

[Abbildung 3: Klassendiagramm Shared-Bibliothek 8](#_Toc446399273)

[Abbildung 4: Klassendiagramm Server 9](#_Toc446399274)

# Einführung

## Aufgabenstellung

Das Modul „VII – Grundlagen Verteilte Informationsverarbeitung“ beschäftigt sich mit der Aufgabe, Datenverarbeitung über die Grenzen eines einzelnen Rechners bzw. eines eng gekoppelten Rechnersystem hinaus zu betreiben. Besonderer Fokus wird dabei auf die Themen „Client-Server-Systeme“ und Remote Procedure Call (RPC) gelegt.

Die Klausuraufgabe für dieses Modul liegt als B-Aufgabe vor, sodass die Aufgabenstellung aus dem Uni-Portal heruntergeladen, die Aufgabe von zu Hause aus bearbeitet und zum Schluss die Lösung über das Uni-Portal eingereicht werden kann.

Die Aufgabenstellung besteht darin, eigenständig ein vollständiges Client-Server-System zu implementieren. Dabei ist der Server parallel zu implementieren, sodass er Anfragen parallel („quasi gleichzeitig“) bearbeiten kann und nicht auf eine sequentielle Bearbeitung („eine nach der anderen“) beschränkt ist. Das konkrete Thema bzw. der Zweck des Client-Server-Systems ist nicht vorgegeben, es dürfen jedoch keine Systeme entwickelt werden, die bereits als Beispiel in den Modulunterlagen auftauchen. Zusätzlich dazu ist eine Dokumentation anzufertigen, die die einzelnen Phasen des Softwareentwicklungsprojekts beschreibt.

Letztendlich sind sowohl der Quellcode des selbst entwickelten Client-Server-Systems, als auch die entsprechend angefertigte Dokumentation einzureichen. Dieses Dokument spiegelt die in der Aufgabenstellung geforderte Dokumentation dar und enthält alle geforderten Elemente.

## Rahmenbedingungen

Die Rahmenbedingungen für das Projekt sind nur relativ grob in der Aufgabenstellung definiert. Die Programmiersprache und Systemumgebung kann in Abstimmung mit dem Dozenten frei gewählt werden. Die eigentliche Implementierung des Systems soll „etwa 10-15 Zeitstunden“ betragen und „3-4 Seiten Code“ ergeben. Natürlich sind diese Angaben relativ zu betrachten: je nach Programmiersprache und verwendeten Bibliotheken ist die identische Funktionalität in der einen Sprache mit wesentlich mehr oder weniger Codezeilen umsetzbar als in einer anderen Sprache. Dazu kommt, dass jede Sprache eine andere Formatierung nutzt [wiki1], sodass der identische Quellcode unterschiedlich formatiert mal mehr, mal weniger Platz benötigt.

Letztlich interpretiere ich diese Angaben so, dass man sich bei dem Projekt nicht in Details verlieren soll, die letztlich nichts mit dem eigentlichen Thema „Client-Server“ zu tun haben. Stattdessen soll der Fokus eben auf genau dieser Thematik liegen.

Als Entwicklungs- und Zielsystem setze ich Microsoft ein. Als IDE (Integrated Development Environment) kommt Visual Studio 2015 Community zum Einsatz, welche kostenlos erhältlich ist [vs1]. Die Laufzeitumgebung ist das .NET-Framework in der Version 4.6, ebenfalls kostenlos erhältlich ist [net1]. Letztlich ist der entwickelte Client und Server auf jedem Betriebssystem lauffähig, auf dem diese Laufzeitumgebung installiert ist (siehe hierzu auch das Projekt „mono“, um .NET unter anderen Plattformen als Windows zu unterstützen [mono1]). Explizit getestet wurde die Software unter den Betriebssystemen Windows 7 und Windows 10. Die Programmiersprache ist C#. Die Dokumentation wird mit Microsoft Office Word 2010 erstellt, zum Schluss jedoch als PDF-Datei exportiert, um die Plattformunabhängigkeit zu gewährleisten.

## Projektbeschreibung

Als Thema für das Client-Server-System möchte ich gern einen rudimentären Webserver inklusive einem Beispielclient implementieren. Die Kommunikation zwischen einem Client (beispielsweise einem Browser) und einem Webserver ist ein klassisches Beispiel für ein Client-Server-System, weshalb dieses Thema der Aufgabenstellung genügen sollte. Da ich beruflich häufig im Webumfeld tätig bin, habe ich zusätzlich dazu auch ein persönliches Interesse daran, die interne Arbeitsweise eines Webservers zu verstehen.

Natürlich kann in diesem Rahmen kein vollwertiger Webserver im Sinne eines Microsoft Internet Information Service (IIS), Apache HTTP Server oder Apache Tomcat implementiert werden. Wichtige Aufgaben für den produktiven Einsatz eins Webservers, wie beispielsweise die Authentifizierung und Autorisierung von Anfragern, Cookie- und Session-Management oder dynamische Programmiersprachen (PHP, JSP, ASP.NET etc.) werden in diesem Projekt nicht berücksichtigt. Der Fokus der Aufgabenstellung – und damit auch des Projekts – liegt in der Implementierung eines Client-Server-Systems, weshalb das Projekt auf diese Kernfunktion beschränkt ist.

Dabei ist jedoch zu beachten, dass der Webserver ein bestimmtes „Feature“ sehr wohl unterstützen muss, nämlich das Protokoll „HTTP“. Dieses Protokoll ist wesentlich für die Kommunikation zwischen einem Client und einem Webserver und ist deshalb auch Bestandteil des dieses Projekts. Allerdings gilt auch hier, dass im Rahmen dieses Projekts nicht die komplette Spezifikation zu HTTP implementiert werden kann, da diese im Detail sehr umfangreich ist [rfc2616]. Auch hierbei wird der Umfang der Implementierung auf das Wesentliche reduziert.

Wichtig ist jedoch, dass am Ende ein grundlegend funktionierender Webserver entsteht: ein Client stellt HTTP-Anfragen für statische Inhalte wie HTML-Dokumente, Javascript- und CSS-Dateien sowie Bilder an den Webserver, dieser nimmt die Anfrage entgegen, sucht in seinem Root-Verzeichnis die Dateien und liefert im sowohl Erfolgs- als auch im Fehlerfall eine sinnvolle HTTP-Antwort an den Client zurück. Dabei arbeitet der Webserver parallel, sodass er eine neue Anfrage entgegennehmen kann, selbst während er eine noch nicht abgeschlossene Anfrage bearbeitet.

# Anforderungsdefinition

## Anforderungen Server

Für den Server soll ein Konsolenprogramm entstehen, das man auf einem Computer mit den entsprechenden Voraussetzungen (siehe Abschnitt „1.2 Rahmenbedingungen“) ausführen kann. Dieses Programm soll auf einen konfigurierbaren Endpunkt (IP-Adresse und Port) hören. Werden Daten in Form eines Bytestreams an diesen Endpunkt geschickt, so sind diese Daten als HTTP-Request zu interpretieren.

Dazu muss die Anfrage entsprechend der Spezifikation [rfc2616-5] geparst werden. Hier ist jedoch anzumerken, dass für den Prototyp ausschließlich die HTTP-Methode „GET“ unterstützt wird. Diese wird dazu verwendet, Ressourcen (insbesondere statische Dateien wie HTML, Javascript / CSS oder Bilder) anzufordern. Alle anderen Methoden – insbesondere POST, PUT und DELETE – erfordern im Regelfall serverseitigen Code zur Bearbeitung der Anfrage (beispielsweise um eine Entität zu speichern, ändern oder zu löschen). Da der Server im Rahmen dieser Projektarbeit jedoch keinen serverseitigen Code unterstützt, können diese HTTP-Methoden ignoriert werden. Sie werden zwar erkannt, jedoch nicht unterstützt. Des Weiteren werden die Request-Header erkannt, haben jedoch ebenfalls keinen Einfluss auf die weitere Verarbeitung.

In jedem Fall beantwortet der Server die Anfrage mit einer validen HTTP-Response. Diese ist ebenfalls entsprechend der Spezifikation [rfc2616-6] aufzubauen. Je nach Art der Antwort ist der entsprechende HTTP-Statuscode zu setzen:

* Ergeben die Daten, die im Server eingegangen sind, keinen validen HTTP-Request, so ist mit dem Statuscode „400 Bad Request“ zu antworten
* Wird eine HTTP-Methode verwendet, die der Webserver nicht unterstützt, so ist mit dem Statuscode „405 Method Not Allowed“ zu antworten
* Wird eine Ressource angefordert, die der Webserver unter dem angegebenen URI nicht finden kann, so ist mit dem Statuscode „404 Not Found“ zu antworten
* Kann die Anfrage korrekt verarbeitet werden, so ist mit dem Statuscode „200 OK“ zu antworten
* Tritt während der Verarbeitung innerhalb des Servers ein unerwarteter Fehler auf, so ist mit dem Statuscode „500 Internal Server Error“ zu antworten

Der Webserver besitzt ein Root-Verzeichnis, in dem die statischen Dateien liegen, die an den Client ausgeliefert werden sollen. Dabei können die Dateien beliebig verschachtelt sein – wichtig ist nur, dass sie sich im definierten Root-Verzeichnis des Servers befinden. Dieses Root-Verzeichnis trägt den Namen „wwwroot“. Außerdem werden lediglich die folgenden Dateiformate unterstützt:

* statisches HTML (.html / .htm)
* Javascript (.js)
* Stylesheets (.css)
* Bilder (.jpg / .jpeg / .gif / .png)

Wird ein valider HTTP-Request entgegen genommen und wird darin eine Datei angefordert, die der Server findet, so öffnet er die Datei, liest den Inhalt aus, und übernimmt ihn in die HTTP-Response.

Zum Schluss wird die Response als Bytestream geparst und an den Aufrufer geschickt.

Die Abbildung auf der folgenden Seite stellt den hier beschrieben schematischen Ablauf nochmals grafisch dar.

Das Programm für den Server soll in der Konsole gestartet werden können, inklusive den Endpunkt-Informationen als Parameterübergabe.

Außerdem soll der Server im parallel-Betrieb arbeiten können. Das bedeutet, dass mehrere Clients gleichzeitig ihre Anfrage an den Server stellen können, und der Server alle Anfragen annimmt und „quasi gleichzeitig“ bearbeitet. Würde der Server sequentiell arbeiten, so könnte er immer nur eine Anfrage nach der anderen entgegen nehmen und bearbeiten – wäre er gerade mit einer Anfrage beschäftigt und würde in diesem Moment eine zweite Anfrage eingehen, würde sie abgewiesen werden, oder im günstigsten Fall in eine Warteschlange aufgenommen werden und von dort sequentiell abgearbeitet (ähnlich der Warteschlange eines Druckers). Natürlich kann ein Client, der gerade eine Anfrage stellen möchte, nicht wissen, wie beschäftigt der Server gerade ist – genauso wenig ist es dem Client zuzumuten, mit seiner Anfrage solange warten zu müssen, bis alle anderen Anfragen abgearbeitet wurden. Aus diesem Grund ist der Server parallel zu konzipieren. Für einen produktiven Webserver wäre es notwendig, die maximal zulässige Anzahl an parallelen Verbindungen konfigurierbar zu machen, da diese Zahl mit der Hardwareausstattung des Rechners und verschiedenen anderen Faktoren abhängt. Für dieses Beispiel genügt es, wenn diese Zahl an einer Stelle fest im Quellcode hinterlegt ist.

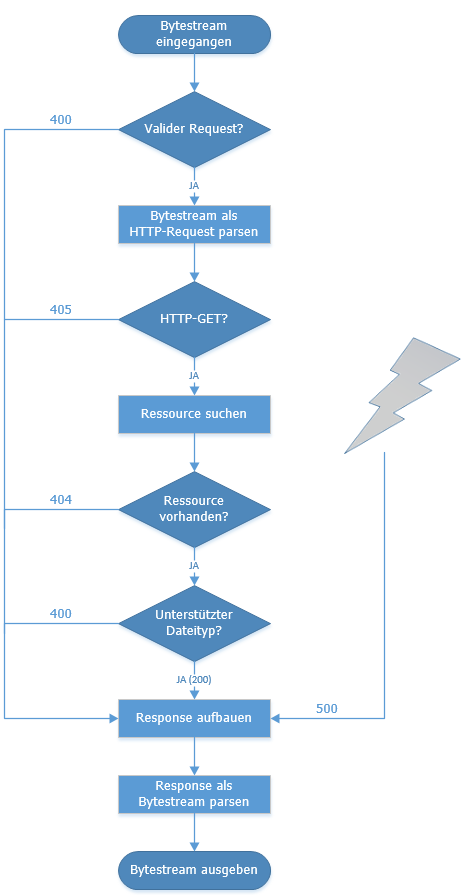


Abbildung 1: schematischer Ablauf AlexWebserver

## Anforderungen Client

Der Client soll ebenfalls ein einfaches Konsolenprogramm sein. Nach dem Start wird der Benutzer aufgefordert, den gewünschten Ziel-URI anzugeben. Hier ist für den Regelfall ein URI auf eine Ressource anzugeben, die unter dem Endpunkt auf dem Server innerhalb des „wwwroot“-Verzeichnisses existiert.

Nach Eingabe des URI ist ein valider HTTP-Request aufzubauen. Dieser soll zur Kontrolle in der Konsole ausgegeben werden und daraufhin an Server geschickt werden. Die vom Server empfangene Antwort ist als HTTP-Response zu parsen und in der Konsole auszugeben.

Dieser Vorgang (URI eingeben, Request anzeigen, Request abschicken, Response anzeigen) soll beliebig oft wiederholt werden können, ohne das Programm neu starten zu müssen.

# Konzeption

## Einleitung

In diesem Abschnitt wird der Webserver konzipiert, das bedeutet in abstrakter Form (ohne Bezug auf konkreten Quellcode o.ä.) beschrieben. Da die Umsetzung in einer objektorientierten Programmiersprache erfolgt, wird für die grafische Darstellung verschiedener Elemente des Konzepts die Unified Modeling Language (UML) verwendet.

## Architekturübersicht

Die grundlegende Struktur des Systems sieht wie folgt aus:

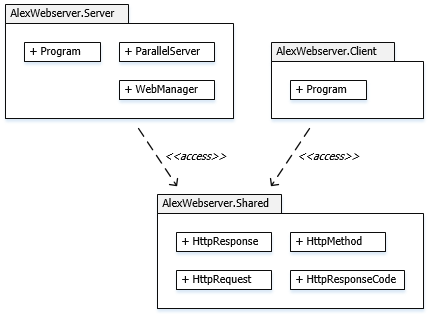


Abbildung 2: Paketdiagramm AlexWebserver

Es gibt eine Komponente für den Server, eine für den Client und eine, die gemeinsam benötige Komponenten enthält.

In dieser „Shared“-Bibliothek finden sich insbesondere Klassen zur Repräsentation eines HTTP-Request, einer HTTP-Response und verschiedene Enumerations, beispielsweise für die HTTP-Methoden (GET, POST, …) und die HTTP-Statuscodes (200, 400, …). Diese Dinge werden sowohl im Server, als auch im Client benötigt, weshalb zur Vermeidung von Redundanzen in eine eigene Komponente ausgelagert werden.

Die Komponente für den Client ist relativ simpel aufgebaut, da hier wenig Funktionalität enthalten ist. Die Funktionalität, die der Client benötigt, ist bereits in der Shared-Bibliothek abgebildet.

In der Komponente für den Server hingegen findet sich etwas mehr Funktionalität, da dieser Aufgaben wahrnimmt, die sonst nirgends benötigt werden.

## Klassendiagramm der Shared-Bibliothek

Die nachfolgende Skizze stellt die grundlegende Struktur der Shared-Bibliothek dar:

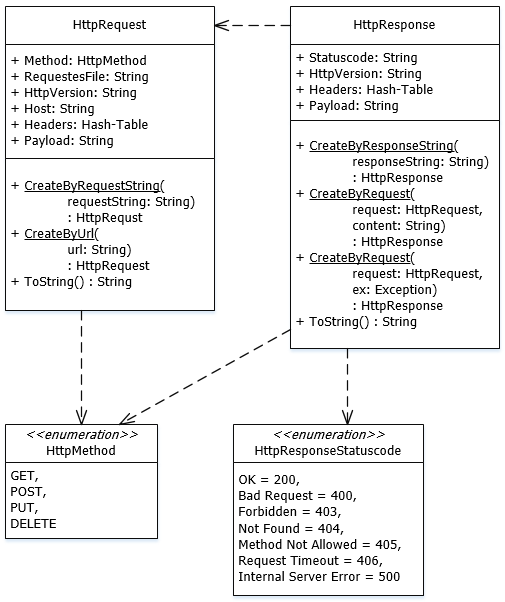


Abbildung 3: Klassendiagramm Shared-Bibliothek

Enthalten sind zum einen Enumerations für HTTP-Methoden und HTTP-Statuscodes. Aus Gründen der Übersicht werden nur die wichtigsten Werte aufgeführt, und nicht alle aus der Spezifikation.

Des Weiteren sind die Klassen für HTTP-Requests und für HTTP-Responses enthalten. Zum einen haben diese Klassen verschiedene Eigenschaften, die an die Werte des jeweiligen Objekts entsprechend der Spezifikation angelehnt sind. Außerdem haben die beiden Klassen verschiedene statische Builder-Methoden („CreateBy…“), um aus verschiedenen Situationen heraus neue Objekte des Typs erstellen zu können. Als Beispiel sei hier die statische Methode „CreateByResponsestring“ der Klasse „HttpResponse“ genannt: als Parameter nimmt diese Methode einen String entgegen, nämlich des als String geparste Byte-Array, des über den Socket aufgenommen wird. Die Methode erzeugt anhand des Strings ein neues Objekt vom Typ „HttpResponse“ und liefert es zurück. Zusätzlich dazu haben beide Klassen eine Methode „Tostring“: diese erzeugt aus dem Objekt wieder einen String mit den Werten des Objekts, der beispielsweise in der Konsole angezeigt werden kann.

## Klassendiagramm Server

Die nachfolgende Skizze stellt die grundlegende Struktur des Servers dar:

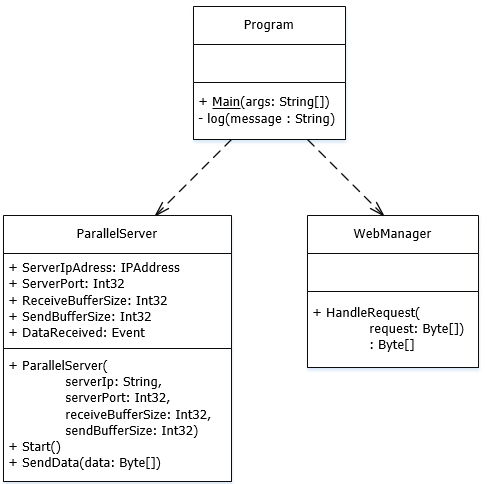


Abbildung 4: Klassendiagramm Server

Neben der Klasse „Program“, die mit ihrer statischen „Main“-Methode das eigentliche Programm startet und der privaten Methode „log“, die einen Text auf der Konsole ausgibt, besitzt die Komponente insbesondere zwei wichtige Elemente.

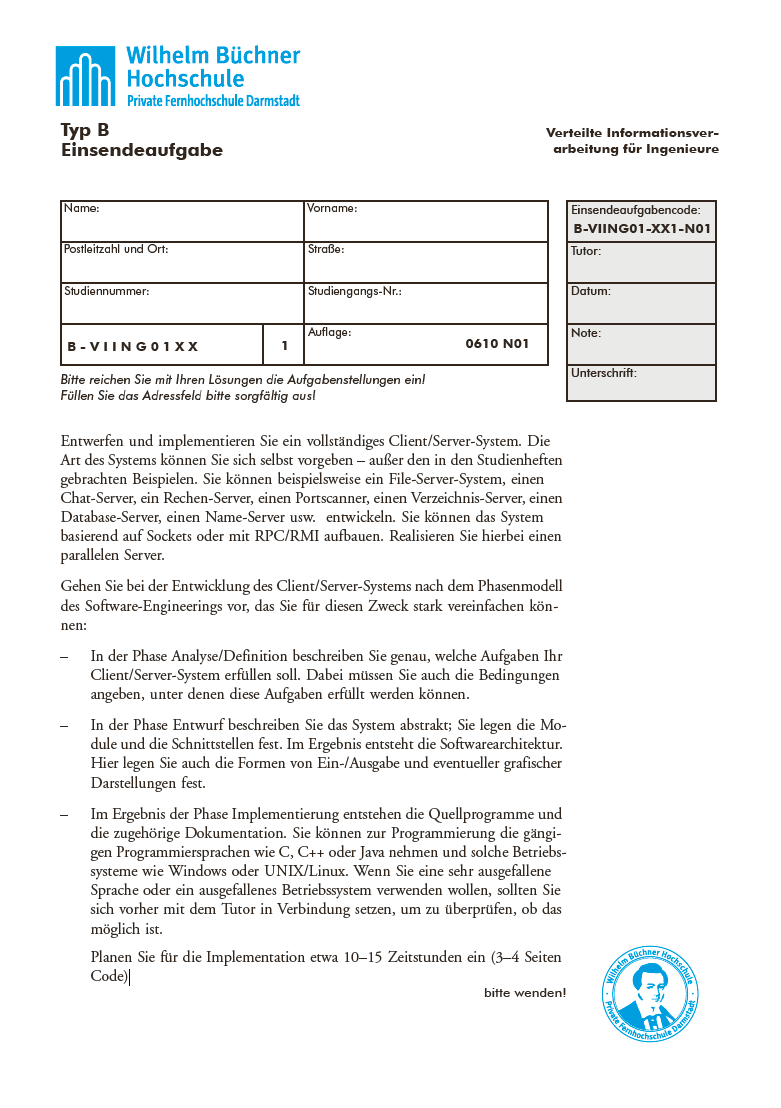
Der Server hat vom Grundsatz her zwei Aufgaben: zum einen muss er aus fachlicher Sicht HTTP-Requests behandeln können, und zum anderen muss er aus technischer Sicht parallel laufen können. Im Sinne einer adäquaten Softwarearchitektur ist es erstrebenswert, diese beiden Komponenten voneinander zu trennen und möglichst lose zu koppeln. Aus diesem Grund existieren in diesem Paket zwei Klassen: eine Klasse „WebManger“, die mit ihrer Methode „HandleRequests“ eine Anfrage in Form eines Byte-Arrays entgegen nimmt und die Antwort ebenfalls in Form eines Byte-Arrays zurückgibt. Diese Klasse hat keinen Bezug zur Technik: weder zum Socket, noch zu einzelnen Threads o.ä. Sie ist für die fachliche Bearbeitung einer Anfrage zuständig. Die andere Klasse „ParallelServer“ ist dafür zuständig, den Socket auf zu machen und beim Empfang von Daten einen neuen Thread zu starten – dies jedoch ohne fachlichen Bezug zu den empfangenen Daten. Das „Austauschformat“ zwischen diesen beiden Klassen ist das Byte-Array – aus Sicht des ParallelServers wird ein Byte-Array vom Socket aufgenommen, und ein Byte-Array wird als Antwort wieder auf den Socket gelegt. Die Interpretation dieses Byte-Arrays ist bereits eine fachliche Aufgabe, die deshalb nicht mehr im ParallelServer, sondern im WebManager abzubilden ist.

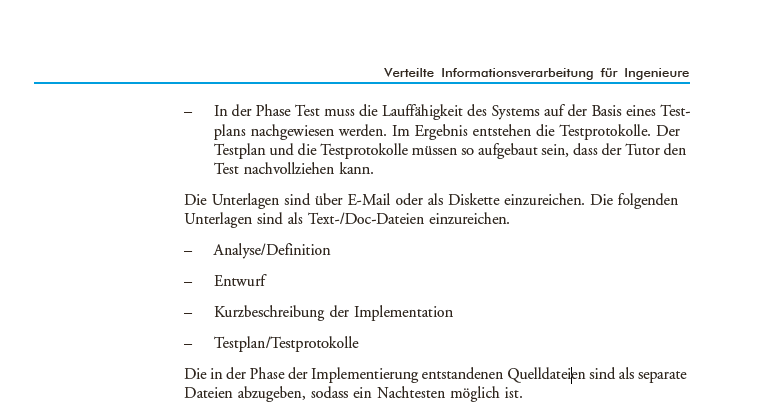
## Tests

Technisch bedeutet dies, dass pro Anfrage ein neuer Thread aufgemacht werden muss bzw. aus einem Threadpool angefordert wird. Der Hauptprozess des Servers ist lediglich dafür zuständig, Anfragen entgegen zu nehmen und diese unverzüglich an einen neuen Thread weiterzuleiten. Die eigentliche Bearbeitung der Anfrage übernimmt der neue Thread.

# Anhang

## Aufgabenstellung





## Quellenangaben

|  |  |
| --- | --- |
| **Kürzel** | **Verweis** |
| wiki1 | <https://de.wikipedia.org/wiki/Einr%C3%BCckungsstil> (21.03.2016) |
| rfc2616 | <https://www.w3.org/Protocols/rfc2616/rfc2616.html> (21.03.2016) |
| vs1 | <https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx> (21.03.2016) |
| net1 | <https://www.microsoft.com/de-DE/download/details.aspx?id=48137> (21.03.2016) |
| mono1 | <http://www.mono-project.com> (21.03.2016) |
| rfc2616-5 | <https://tools.ietf.org/html/rfc2616#section-5> (21.03.2016) |
| rfc2616-6 | <https://tools.ietf.org/html/rfc2616#section-6> (21.03.2016) |