# Verteilte Systeme

Design der HTTP-Server/Container

Die im Paket *de.htw.ds.http* enthaltenen Klassen konstituieren Server welche über HTTP mit Clients kommunizieren. Clients können dabei entweder Web Browser beliebiger Art, oder z.B. die enthaltene Klasse *HttpClient* sein.

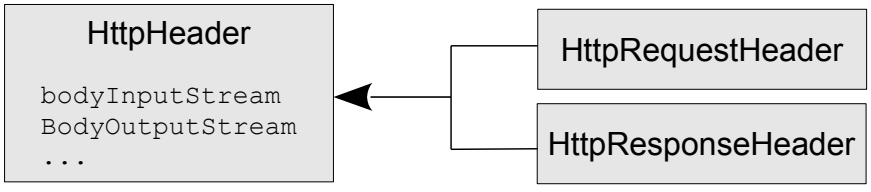
## *HTTP-Protokoll*

HTTP ist ein gemischt auf Text und Binärdaten basierendes Protokoll. Über eine vom Client initiierte TCP-Verbindung wird dabei eine Anfrage (HTTP-Request) an den Server gesendet, auf welche dann vom Server-Gegenstück über die gleiche TCP-Verbindung eine Antwort (HTTP-Response) gesendet wird. Eine HTTP-Anfrage besteht dabei aus einem textbasierten HTTP-Request-Header, auf welchen der Inhalt optionaler binärer Dokumente in einem HTTP-Request-Body folgen kann. Eine HTTP-Response besteht ebenfalls aus einem textbasierten HTTP-Response-Header, auf den optional der Inhalt eines binären Dokuments folgen kann. Text-Header und binäre Dokumente sind dabei durch eine Leerzeile (i.e. zwei aufeinander folgende Zeilenumbrüche) getrennt.

Ein HTTP-Client darf gleichzeitig mehrere Verbindungen zu einem HTTP-Server öffnen um Anfragen parallel beantworten zu lassen. Unter gewissen Umständen ist es zudem möglich über eine TCP-Verbindung mehrere Anfrage/Antwort Paare zu kommunizieren. Die immer noch aktuelle HTTP-Version 1.1 ist in RFC2616 beschrieben, in Kurzform mag jedoch eine EBNF-Grammatik zum Verständnis genügen:

```
HttpConnection := { HttpRequest HttpResponse }
HttpRequest := HttpRequestHeader { Document }
HttpResponse := HttpResponseHeader [ Document ]
HttpRequestHeader := RequestLine { Property } CR
HttpResponseHeader := StatusLine { Property } CR
RequestLine := Method " " Path " HTTP/" Version "." Revision CR
StatusLine := "HTTP/" Version "." Revision " " Code " " Comment CR
Property := Name ": " Value CR
Name := String
Value := String
Document := { byte }
Method := "GET" | "POST" | "HEAD" | "PUT" | "DELETE" | "TRACE" | "OPTIONS" | "CONNECT"
Path := String
Version := "1"
Revision := "0" | "1"
Code := digit digit digit
Comment := String
CR := newline
```

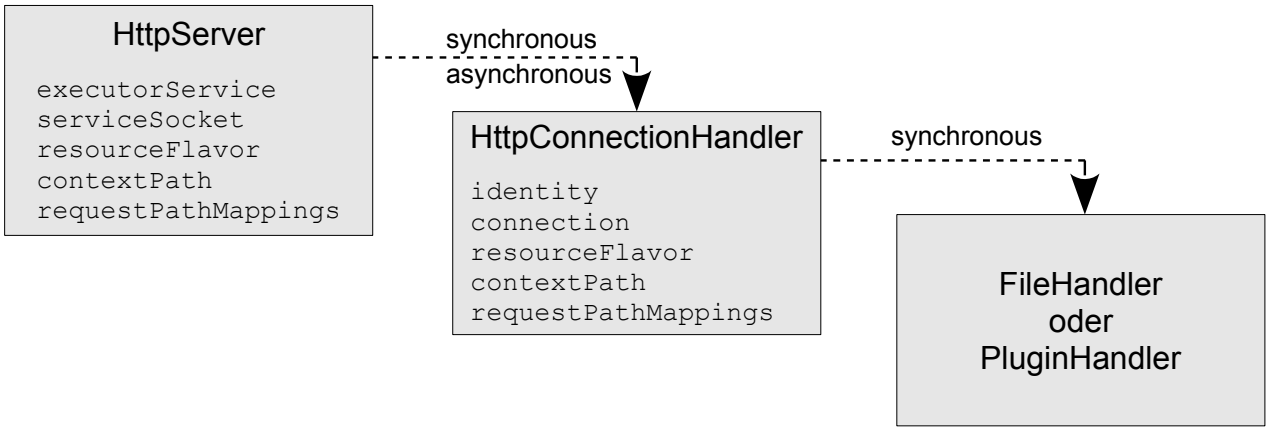
Die Klassen *HttpRequestHeader* und *HttpResponseHeader* modellieren die über eine HTTP-Verbindung kommunizierten Daten. Die Klassen verfügen zudem über r*ead()* bzw. *write()* Methoden zum Parsen bzw. Generieren der textbasierten Header. Beachtet dass die abstrakte Superklasse *HttpHeader* die Gemeinsamkeiten im Design der beiden Klassen abbildet.



Zusätzlich verfügen die Header-Klassen über assoziierte innere Stream-Klassen: Ein *HeaderInputStream* ist in seiner Lesefähigkeit auf die im Parent-Header definierte Content-Länge begrenzt. Ein *HeaderOutputStream* dagegen schreibt die Daten des Parent-Headers zum spätmöglichsten Zeitpunkt auf den unterliegenden Ausgabe-Datenstrom. Damit wird ermöglicht dass Server-Komponenten noch zu einem relativ späten Zeitpunkt Details des Headers (wie den Return-Code) setzen und an den Client versenden können. Instanzen dieser Stream-Klassen können aus HTTP-Headern mittels `getBodyInputStream()` bzw. `getBodyOutputStream()` abgefragt werden.

## *Server-Design*

Alle Http-Server folgen demselben generellen Design:




Alle Http-Server Klassen sind als *Runnables* ausgeführt, und blockieren in der run()-Methode bis eine Client-Verbindung eingeht. Ebenfalls gemeinsam ist allen Http-Servern dass sie eingegangene Verbindungen zur Verarbeitung an Instanzen von *HttpConnectionHandler* weitergeben. Diese Handler wiederum entscheiden aufgrund eines *ResourceFlavors* ob Ressourcen statisch oder dynamisch zu Handhaben sind, und entscheiden letztlich auch welche Art *HttpRequestHandler* mit der Abarbeitung einer Anfrage betraut wird.

Instanzen der Klasse *HttpServer1* und *HttpServer2* setzen dabei *StaticResourceFlavor*-Instanzen ein welche ausschließlich statische Datei-Ressourcen unterstützen. *HttpServer1* implementiert dabei das statische Server-Entwurfsmuster indem der Aufruf des Connection-Handlers synchron erfolgt, während *HttpServer2* das dynamische Server-Entwurfsmuster mittels asynchronem Aufruf realisiert. Dieser Connection-Handler wiederum beauftragt für jede Anfrage eine *FileHandler*-Instanz synchron mit der Beantwortung. Zudem implementieren alle Connection-Handler das HTTP 1.1 Verbindungs-Caching, sowie Anfragepfad-Mapping.

Instanzen der Klasse *HttpContainer1* dagegen leiten eingehende Verbindungen immer asynchron an Instanzen von *HttpConnectionHandler* weiter, und verwenden daneben Instanzen von *DynamicResourceFlavor*. Letztere ermöglichen es, neben Anfragen nach statischen Ressourcen (mittels *FileHandler*) auch dynamisch pro Anfrage erzeugte Ressourcen mittels Instanzen von *PluginHandler* zu adressieren. *DynamicResourceFlavor* stellt den Plugins beim Prozessieren zudem ein Context-Objekt zur Verfügung welches erweiterte Variablen-Scopes (Global, Session, Request) enthält.

Einzige Voraussetzung dazu ist dass die mittels URL adressierten Plugin-Klassen das *HttpRequestHandler* Interface implementieren, und zudem einen öffentlichen Default-Construktor zur Verfügung stellen. Es genügt deren java- oder class-Files im Context-Directory des Containers abzulegen - inklusive Verzeichnis-Hierarchie zur Abbildung der Java-Pakete. Der Plugin-Handler kann dann die Plugin-Klassen aus Java Kompilaten (.class) mittels Java Reflection API dynamisch laden und instantiieren, ohne dass diese beim Container-Start per ClassPath erreichbar sein müssen; dazu wird ein temporärer Thread verwendet dessen Context-Classloader vor dem Start passend eingerichtet wird. Bei Quellcode (.java) erfolgt zuvor noch eine Kompilation falls dieser aktueller als ein eventuell vorhandenes Kompilat ist.

Der Aufruf einer solchen dynamischen Ressource kann dann im Browser z.B. folgendermaßen aussehen: `http://localhost:8002/de.htw.ds.http.handler.MemoChat.java`. Einige Plugin-Klassen finden sich im JAR-File *code/distributed-systems-plugins.jar*, und können dort mittels Unzipper als Quellcode und/oder bereits kompiliert extrahiert werden.