Java Servlets

Eine Einführung

Angefertigt von

Marion Beckers

im Proseminar *„Technologien zur Entwicklung von Webanwendungen“*
Lehrstuhl III für Informatik
Sommersemester 2002

RWTH AACHEN

Inhaltsverzeichnis

1 Motivation

1.1 Einführung

Die Motivation, Servlets zu entwickeln entstand aus dem Streben nach dynamischen HTML-Dokumenten. Dynamisch bedeutet dabei, daß beiderseitige Kommunikation - also zwischen Client und Server möglich ist - so wie es zum Beispiel bei datenbank-generierten Seiten, Auskunftssystemen oder Shopping-Systemen der Fall ist. Entsprechend gab es vor dieser Entwicklung nur statische Datenübermittlung.
1993 wurde dazu das erste mal das Common Gateway Interface (CGI) benutzt, welches 1995 standarisiert angeboten wurde. Allerdings wurde 1996 aufgrund von Defiziten, die CGI mit sich brachte, erstmalig Java benutzt, um serverseitige Erweiterungen zu programmieren. Anfang 1997 wurden diese Servlets, welche bis dahin bei den einzelnen Webservern unter verschiedenen Namen liefen, zu einem Standard vereinheitlicht. Im August 1997 gab es dazu schließlich den ersten Servlet-Container für den Apache Webserver mit Namen JServ (vgl.[koel99]).
Heute sind Servlets aufgrund der vielseitigen Möglichkeiten, die sie bieten, nicht mehr ersetzbar. Das Proseminar "Java Servlets - Eine Einführung" widmet sich den Funktions- und Programmiergrundlagen dieser Servlets speziell in Bezug auf das HTTP- Protokoll und soll erste Schritte zur eigenen Entwicklung weisen.

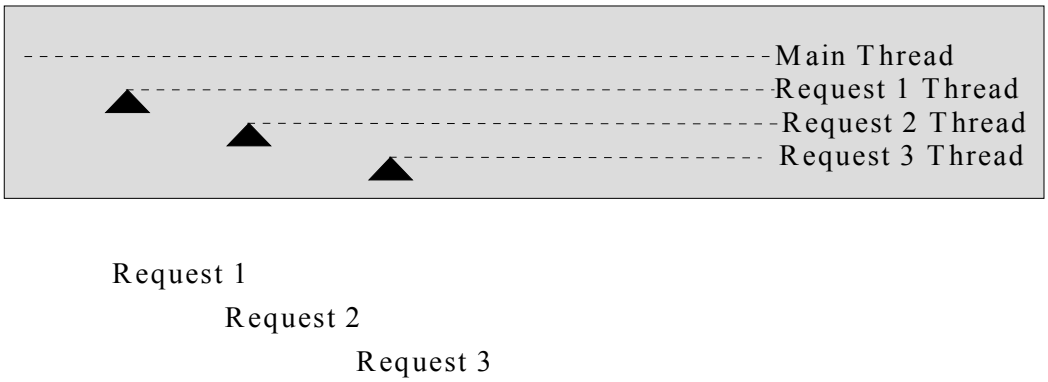
1.2 Was sind Servlets?

Servlets sind serverseitige, dynamische Softwarekomponenten, die es erlauben, die Funktionalität des Servers zu erweitern. Durch diese Erweiterungen ist es möglich, daß der Klient Einfluß auf das erwartete und vom Server gegebene Ergebnis hat. Dadurch, daß sie in Java geschrieben werden, können sie programmiertechnisch als gewöhnliche Java-Klassen angesehen werden, auch wenn der Umgang mit den speziellen Methoden zunächst unbekannt erscheint.
Die Voraussetzung zum Entwickeln und Benutzen von Java Servlets ist neben der für alle Java-Programme benötigte Java 2 Standard Edition (J2SE) beziehungsweise dem Java Development Kit (JDK) das Java Servlet Development Kit (JSDK). Dieses beinhaltet unter anderem das Servlet Application Programming Interface (API), eine spezielle Kollektion von Klassen und Interfaces, die für die Erstellung und Ausführung von Servlets notwendig sind. Die Servlet API ist alternativ in der Java 2 Enterprise Edition (J2EE) enthalten, es besteht aber auch die Möglichkeit, diese separat herunterzuladen (siehe hierzu [java02] und [serv01])
Die Aufgabe eines Servlets ist es, vom Server weitergeleitete Anfragen (Requests) auszuwerten und die dazu gehörigen Antworten (Responses) zurückzugeben. Dies geschieht durch Aufbauen einer entsprechenden HTML-Datei. Die Request-Response Aufgabe, durch die 2-Wege-Kommunikation stattfinden kann, findet ihre Anwendung in einer Vielzahl von Fällen. Einige davon werden später in Kapitel 2.1 vorgestellt.

1.3 Vorteile von Servlets

Die Anwendung von Servlets ist aus vielen Gründen vorteilhaft. So spielt die Tatsache der Unabhängigkeit von Servlets eine große Rolle. Servlets sind zum einen plattform- und serverübergreifend, zum anderen aber auch protokollunabhängig (SMTP, POP3, HTTP...). Die Plattformunabhängigkeit erreichen Servlets dadurch, daß sie in Java geschrieben sind. So können sie ohne Modifikationen auf verschiedenen Plattformen (wie zum Beispiel Unix, Windows,...) verwendet werden. Ebenso laufen sie auf allen bekannten Webservern wie auch FTP-, Telnet-, Mail- und News-Servern.
Als weiterer Vorteil ist die Möglichkeit zu nennen, alle notwendigen Ressourcen in eine Datei packen zu können. Die sogenannte "Web-Application", eine Kollektion von Servlets, Java Server Pages, HTML-Dokumenten und allen anderen Ressourcen besitzt die Endung `.war` und beinhalten eine Datei (das "deployment descriptor file"), welches Bestandteil der Applikation ist. Diese instruiert den Server, wie die Applikation zu installieren ist. Das Erstellen eines Servlets wird so entsprechend erleichtert.
Servlets garantieren desweiteren weitestgehend Sicherheit. "Weitestgehend" bedeutet, daß natürlich keine absolute Sicherheit garantiert werden kann. Sicherheit beinhaltet zum einen die Sicherheit vor einem Absturz durch Einsatz einer Java Virtual Machine (JVM). Diese führt das Servlet aus und stellt sicher, daß Servlets keinen Crash durch unerlaubten Zugang zum Datenspeicher verursachen können, weil die JVM keinen Zugang zu diesem erlaubt. Außerdem ruft die JVM eine Ausnahmebehandlung auf, wenn Fehler entdeckt worden sind, anstatt einen Crash zuzulassen und verifiziert, daß kompilierte Java-Klassen keine illegalen Operationen aufrufen können.
Ein anderer Vorteil der Benutzung einer solchen JVM ist, daß Servlets bei der Kompilierung in Java-Bytecode überführt werden. Der Bytecode ist quasi Maschinencode für die JVM. Durch den Vorgang des Kompilierens können Syntaxfehler erkannt und die benutzten Typen auf ihre Richtigkeit überprüft werden. So werden Servlets stabiler und sicherer. Die Kompilierung in Bytecode hat den Vorteil, daß die Servlets schneller in der Ausführung werden.
Zum anderen wird Sicherheit vor ungewolltem Datenzugang gewährleistet, einerseits durch den Gebrauch des "Security Managers" des Servers, andererseits durch die Möglichkeit der Authentifizierung und Autorisierung der Benutzer. Das Servlet hat bei jeder weiteren Anfrage Kenntnis der Identität des anfragenden Clients.
Es ist weiterhin möglich, Servlets sowohl lokal, als auch von einem anderen Server im Netzwerk zu laden. Dies erspart die Arbeit, jedes Servlet auf jeden eventuell benutzten Server zu kopieren. Der Vorteil ist, daß die Servlets keine Ressourcen dieser Server verbrauchen und nur geladen werden, wenn sie auch wirklich gebraucht werden.
Dadurch Servlets im selben Prozess laufen wie der Server wird die Servlet-Server-Kommunikation vereinfacht. Das Servlet kann auf gewisse Serverressourcen zugreifen und muß nur einmal beim erstem Gebrauch oder beim Starten des Servers geladen werden. Einmal geladen verweilen sie im Speicher bis zum Aufruf der `destroy`-Anweisung. Die Methode `destroy` wird aufgerufen, wenn das Servlet nicht mehr gebraucht wird oder wichtige Ressourcen freigegeben werden müssen. Bis dahin ist es möglich, daß das Servlet mehrere Anfragen bearbeiten kann.
Anfragen werden durch sogenannte Threads vom Servlet behandelt. Servlets sind fähig, mehrere Threads in einem Prozeß zu verarbeiten. Dadurch wird jede Anfrage als Thread innerhalb des Webserver-Prozesses realisiert, (Servlets laufen in dem selben Prozeß wie

der Server) wie in Abbildung 1 gezeigt wird. Der Kasten verdeutlicht den einen Prozeß (Vgl. [call00] und Abbildung 2, Vorgehensweise bei CGI-basierten Servern)

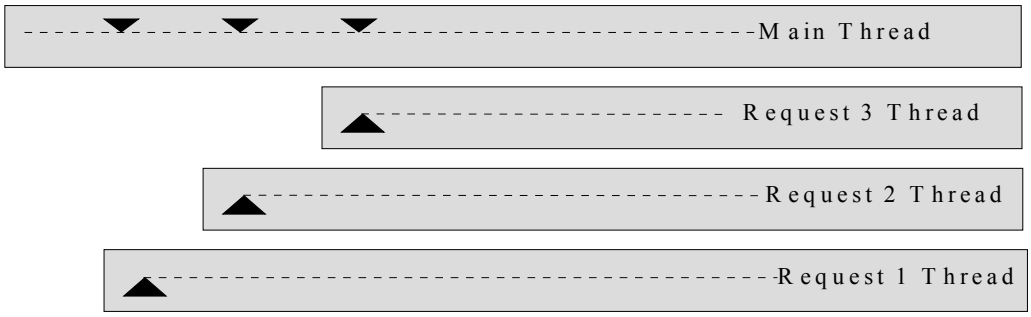


[Abb. 1: Java-basierter Server]

Es ist allerdings möglich, aus Gründen der Threadsicherheit durch das Interface `javax.servlet.SingleThreadModel` mehrere Servlet-Instanzen zu erzeugen, so daß jede Instanz nur einen Request gleichzeitig behandeln kann. Diese Vorgehensweise soll aber im weiteren Verlauf vernachlässigt werden.

1.4 Alternativen und Vergleiche

Zu Servlets konkurrierende Web-Applikationen sind unter anderem CGI, FastCGI, NSAPI und ISAPI. Diese vier Applikationen sollen hier vorgestellt und diskutiert werden.
Das Common Gateway Interface (CGI), das bereits in Kapitel 1.1 vorgestellt wurde, schneidet im Vergleich zu Servlets aus folgenden Gründen deutlich schlechter ab: CGI ist unter Umständen nicht plattform- oder serverunabhängig. CGI-Anwendungen müssen für jeden Request einen neuen Prozeß starten, ist als nicht multithread-fähig. Diese Prozeßerstellung kostet sowohl Zeit als auch Serverressourcen. Der Unterschied zwischen der Vorgehensweise von Servlets und von CGI wird deutlich, wenn man die Abbildung 2 mit der Abbildung 1 vergleicht. Die Kästen stellen auch hier wieder die Prozesse dar.



[Abb. 2: CGI-basierter Server]

Da kommt hinzu, daß CGI-Anwendungen, falls sie in C geschrieben werden, fehleranfälliger als Servlets sind, da Servlets durch die JVM wie oben beschrieben "kontrolliert" werden und auf den Java Security Manager zurückgreifen können. Als positiver wie auch negativer Aspekt kann gedeutet werden, daß CGI nicht im selben Prozeß abläuft wie der Server. Dadurch wird CGI-Programmen einerseits zwar die Möglichkeit genommen, auf die Serverressourcen zurückzugreifen, andererseits schützt Prozeßisolation den Server vor einem Crash durch Fehler im CGI-Prozeß.
FastCGI ist eine Weiterentwicklung von CGI der Firma Open Market. Es ist eine Kombination aus CGI und API's. So konnte es effizienter gestaltet werden. Als wesentliche Änderung ist Persistenz zu nennen. Das bedeutet, daß vom Server ein Pool von FastCGI-Prozessen erzeugt wird, welche dauerhaft laufen, sich also nicht nach dem Behandeln eines Requests beenden. Ein FastCGI-Prozess kann allerdings immer noch nicht mehrere Threads gleichzeitig behandeln, was dazu führt, daß bei einer hohen Anzahl gleichzeitiger Requests lange Wartezeiten entstehen können.
Die Netscape Server API (NSAPI) und die Internet Server API (ISAPI) sind Entwicklungen der Firma Netscape beziehungsweise Microsoft. Sie sind in ihrer Leistung vergleichbar oder besser als Servlets und übertreffen CGI wesentlich. Diese API's laufen im Serverprozeß, was die bereits angegebenen Vor- und Nachteile nach sich zieht. Die größten Nachteile von NSAPI und ISAPI sind jedoch, daß sie zum einen komplexer sind, woraus höhere Implementierungs- und Wartungskosten entstehen, zum anderen sind sie nicht server- und plattformübergreifend. So ist NSAPI nur mit wenigen Webservern kompatibel, welche von Netscape angeboten werden, und ISAPI benötigt für gewöhnlich den Microsoft Internet Information Server (IIS) auf Windows NT/2000. Beide müssen überdies für jede Plattform neu kompiliert werden (vgl. [call00]).

2 Anwendungs- und Leistungsumgebung

2.1 Anwendungsumgebung und Leistungsumfang

Servlets wurden entwickelt, um, ähnlich wie CGI, Webseiten dynamisch gestalten zu können. Dabei geht es vor allem darum, Server und Client wechselseitig miteinander in Verbindung zu setzen, wie es zum Beispiel bei jeder Art des e-commerce der Fall ist. Ein Servlet erlaubt dem Server, mit dem Applet des Clients zu kommunizieren. Weiterhin sind Servlets in der Lage, Anfragen entgegenzunehmen, den Input auszuwerten und eine entsprechende Antwort (in Form einer HTML-Datei) zurückzuliefern. Desweiteren soll Kommunikation auch zwischen mehreren Mitgliedern gleichzeitig möglich sein, wie es beispielsweise bei interaktiven Computerspielen der Fall ist.
Servlets sind außerdem fähig, Requests zu anderen Servern weiterzuleiten, einen Service zu unterteilen und mit den Serverresourcen zusammenzuarbeiten, um effizientere Aufgabenbewältigung betreiben zu können. Durch Abfragen (etwa von Benutzername und Kennwort) bestimmter Informationen können Benutzer authentifiziert werden. Den Nutzen dieser Leistung trifft man zum Beispiel bei Postfach-Anbietern. Bei neuen Anfragen desselben Nutzers ist das Servlet in der Lage, die angegebenen Informationen zu speichern und, falls erforderlich, die Autorisierung erneut zu überprüfen (vgl. [koel99]).

2.2 Laufzeitumgebung

Damit ein Servlet ausgeführt werden kann, stellen sogenannte Container eine Laufzeitumgebung für Komponenten zur Verfügung. Anfragen des Benutzers werden durch den Container gesteuert und an die Java Virtual Machine weitergeleitet. Der erste Servlet-Container wurde 1997 verabschiedet und trägt den Namen JServ. Der Nachfolger von JServ wurde später von Sun/Apache entwickelt und heißt Catalina. Der heute bekannteste Servlet-Container ist Tomcat. Die erste Version (Version 3.0) von Tomcat wurde im Dezember 1999 unter dem Projektnamen Jakarta fertiggestellt. Servlet-Container können grob in 3 Gruppen eingeteilt werden (vgl. [raas01]):

1. Der Container ist Bestandteil des Servers, wie es bei Java-basierten Servern der Fall ist ("stand-alone servlet container").

2. Es existiert eine Kombination aus Web-Server-Plugin und einer Java Container Implementierung. Der Container läuft hier in der JVM ab, die im Adreßbereich des Webservers geöffnet wird. Das Plugin hat dabei die Aufgabe, die Requests anzunehmen und übergibt diese dann an den Container ("in-process servlet container").

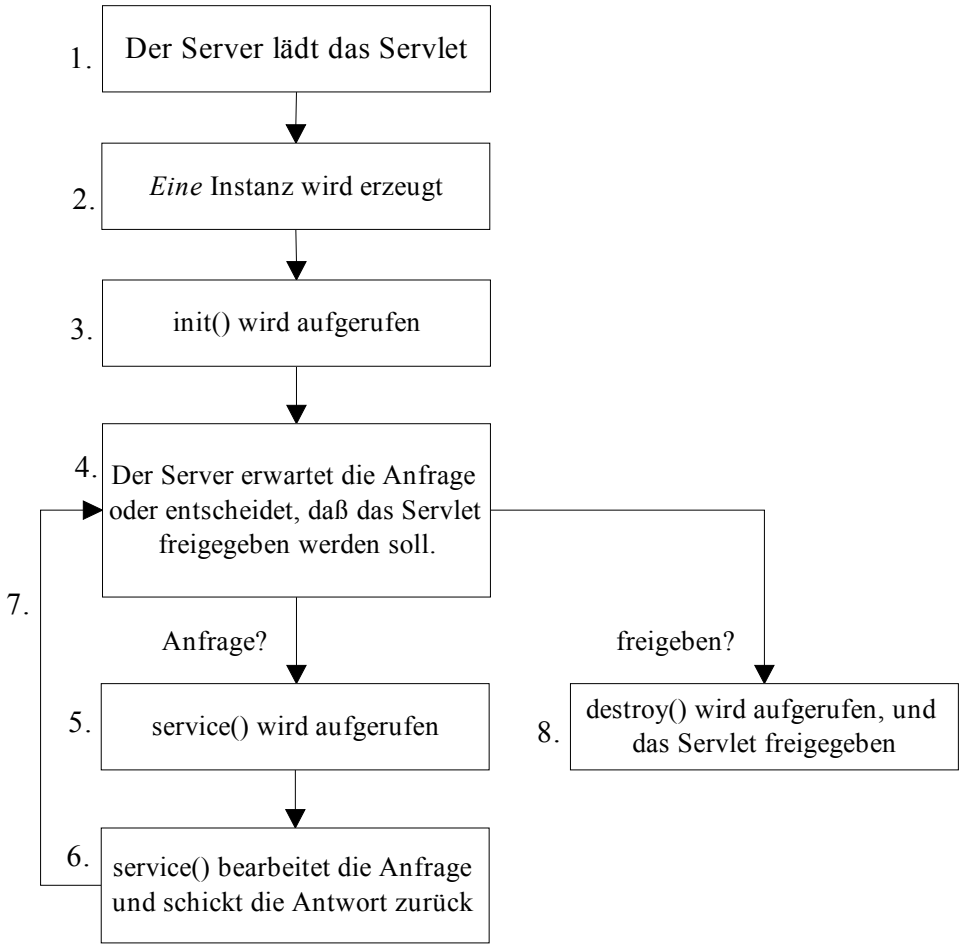
3. Die Container laufen in einer externen JVM ab. Wiederum übernimmt das Plugin die Requests und leitet sie weiter an den Container. Die Vorteile liegen in der Stabilität dieser Konstruktion, die Nachteile in der mangelnden Schnelligkeit und damit mangelnden Performanz der Übermittlung ("out-of-process servlet container")

Container sind eine Notwendigkeit zur Ausführung von Servlets. Sie sind allerdings nur bedingt plattform- und serverunabhängig, da ein Container sich auf den entsprechenden Server beziehen muß. Ein übergreifender Container ist zum Beispiel JRun. Er ist fähig, auf dem Apache Webserver, aber auch auf einer Reihe anderer Server zu laufen, wie beispielsweise dem Microsoft Personal Webserver 4.x, Microsoft Internet Information Server 3.x/4.x, Netscape FastTrack und Enterprise Server 3.x oder StartNine WebSTAR 3.x. Auch Tomcat ist serverübergreifend. Die Version 3.2.4 kann eingesetzt werden für den Apache Web-Server ab Version 1.3, aber auch für Microsoft- oder Netscape-Server. (vgl. [raas01], [jaka02]). Tomcat ist erhältlich unter [jaka02].

3 Grundlagen der Programmierung

3.1 Prinzipielle Ablaufstruktur

Der Lebenszyklus eines Servlet-Prozesses wird vor allem bestimmt durch die Methoden init() und destroy(). In Abbildung 3 wird zusätzlich der Zusammenhang von Server zu Servlet, also der Ladevorgang vor der init()-Methode gezeigt.



[Abb. 3: Lebenszyklus eines Servlets]

Der Prozeß bis zum endgültigen "zerstören" des Servlets ist in die von 1 bis 8 nummerierten Schritte zerlegbar:

1. Der Server lädt das Servlet beim ersten Bedarf oder alternativ beim Starten des Servers. Das Laden "bei Bedarf" kann entweder lokal oder von von einem anderen Server im Netzwerk aus geschehen.

2. Der Server erstellt eine Instanz des Servlets, um die verschiedenen Requests bearbeiten zu können. Durch die Multithread-Fähigkeit ist nur eine Instanz notwendig. (siehe auch Kapitel 1.3, SingleThreadModel)

3. Die `init`-Methode wird aufgerufen. Alle für die Initialisierung notwendigen Parameter sind in dieser Methode vorhanden oder werden an diese übergeben.

4. Wenn den Server ein Request erreicht, erzeugt dieser ein `HttpServletRequest`-Objekt sowie ein entsprechendes `HttpServletResponse`-Objekt. Wird alternativ durch den Server (bzw. Servlet-Container) entschieden, daß das Servlet nicht mehr in Gebrauch sein sollte, wird bei Punkt 8 fortgesetzt. Diese Entscheidung

kann getroffen werden, wenn kein Bedarf mehr besteht oder das Servlet benötigte Ressourcen blockiert.

5. Sobald ein Request eingegangen ist, wird die im Servlet befindliche `service`-Methode aufgerufen. Bei gleichzeitigem Eingehen mehrerer Requests wird die `service`-Methode in der entsprechenden Anzahl aufgerufen.

6. Durch das `HttpServletRequest`-Objekt erreicht der Request die Methode `service` und kann von dieser bearbeitet werden. Die erzeugte Antwort wird durch das `HttpServletResponse`-Objekt an den Klienten zurückgebeben.

7. An dieser Stelle ist die Aufgabe des Servlets abgeschlossen. Der Server springt zurück an Punkt 4 und trifft abhängig von eingehenden Requests erneut die Entscheidung, ob das Servlet zerstört werden soll. Gehen weitere Requests ein und das Servlet soll nicht aus anderen Gründen zerstört werden, wird Punkt 5 erneut aufgerufen.

8. Die `destroy`-Methode wird aufgerufen. Gebrauchte Ressourcen werden freigegeben, das Servlet kann jetzt vom Garbage Collector endgültig zerstört werden (vgl. [call00])

3.2 Servlet Reloading

Der Prozeß des "Servlet Reloading" eröffnet die Möglichkeit, das Servlet neu zu kompilieren und zu testen, ohne das der Server neu gestartet werden muß. Dies ist vor allem in der Entwicklungsphase wichtig, aber auch wenn eine neue Version eines Servlets benutzt werden soll, da der Betrieb nicht unterbrochen werden muß. Manche Server laden zusätzlich alle Klassen neu, die sich im selben Verzeichnis wie das Servlet befinden. Ermöglicht ein Server diesen Vorgang nicht, ist es notwendig den Server neu zu starten, um entsprechende Änderungen übernehmen zu können.
Die Nachteile des "Servlet Reloading" sind ein eventueller Leistungsverlust und eine gewisse Fehleranfälligkeit durch die Verwendung der neugeladenen Klassen.

3.3 Methodischer Aufbau

Bevor das Programmierbeispiel behandelt wird, soll ein prinzipieller Aufbau gezeigt werden. Jedes Servlet besteht aus Java-Klassen, die alle entweder die Klasse `GenericServlet` oder `HttpServlet` erweitern. In der Servlet API ist das Paket `javax` zu finden, welches unter anderem die Pakete `javax.servlet` und `javax.servlet.http` beinhaltet. Diese Pakete beinhaltet die Klassen `HttpServlet, welche` ihrerseits die Klasse `GenericServlet` erweitert. `HttpServlet` beinhaltet die nun vorgestellten Methoden (siehe Abbildung 4). Die Methoden `init`, `destroy` und `service`, welche in den vorangegangenen Kapiteln schon mehrfach erwähnt wurden, sind für Servlets von großer Wichtigkeit, obwohl es theoretisch möglich ist, diese nicht zu implementieren. Jedes Servlet überschreibt zumindest eine der gegebenen Methoden,

für gewöhnlich jedoch die Methoden innerhalb von `service()`. Das Skelett eines Servlets sieht damit wie folgt aus:

```
public class SkeletonServlet extends HttpServlet
{
    public void init() throws ServletException {}

    public void service() throws ServletException,
    IOException {}

    public void destroy() {}
}
```

[Code 1: Skelett eines Servlets]

Die Aufgaben der Methoden sind wie folgt (vgl. [call00] und [koel99]):

- **`init()`**

Die Methode `init()` wird genau einmal aufgerufen. Dies geschieht zum Zeitpunkt des Ladens. Da das Servlet im Speicher verweilt, bis es endgültig zerstört wird, muß die Methode kein zweites Mal aufgerufen werden. Sie enthält alle notwendigen Initialisierungsparameter und entspricht somit dem Konstruktor. Enthält `init()` Objekte, welche die `service`-Methode ebenfalls benutzt, muß sogenannte Threadsicherheit gewährleistet werden, da es `service()` möglich ist, gleichzeitig mehrere Requests zu behandeln und sie damit gleichzeitig aufgerufen werden kann. Threadsicheres Programmieren kann durch das `SingleThreadModel` umgangen werden.
Die Methode `public void init() throws ServletException` kann im Fall eines Fehlers eine Ausnahmebehandlung zurückzuliefern. Fehler sollten immer vom Servlet mit einer Ausnahmebehandlung bearbeitet werden. Geschieht dies nicht und `init()` würde die `System.exit`-Methode aufrufen, kann es passieren, daß die JVM und damit der Container zerstört wird.
Neben der parameterlosen Version der `init`-Methode existiert parallel die Methode `init(ServletConfig config)`.Diese diente ursprünglich dazu, Zugang zu dem `ServletConfig`-Objekt zu bekommen. Die parameterlose Version ist allerdings vorzuziehen, denn es gibt heutzutage keinen Grund mehr, das `ServletConfig`-Objekt so erreichen zu wollen. Der Grund ist, daß das `ServletConfig`-Objekt durch die Methode `getServletConfig` erreicht werden kann, wenn es benötigt wird.

- **`destroy()`**

Die `destroy`-Methode wird aufgerufen, wenn das Servlet nicht mehr gebraucht wird oder wenn es Ressourcen blockiert, die benötigt werden. Das Servlet kann dann keine Requests mehr empfangen. Bevor `destroy()` allerdings aufgerufen werden kann, werden erst alle anstehenden Requests verarbeitet. Die Methode sollte benutzt werden, um Ressourcen freizugeben, bevor das Servlet endgültig zerstört wird. Es kann dann vom Garbage Collector übernommen werden. Damit beendet `destroy()` den Lebenszyklus des Servlets. Der Aufruf dieser Methode wird für gewöhnlich vom Server übernommen.

- **`service()`**

Erreicht, wie in Kapitel 3.1 beschrieben, den Server ein Request, baut dieser ein `HttpServletRequest`-Objekt und ein `HttpServletResponse`-Objekt auf. Diese Objekte sollten der `service`-Methode des Servlets übergeben werden durch den Aufruf von `service(HttpServletRequest request, HttpServletResponse response)`. Wie auch bei `init()` sollte `service()` Ausnahmen abfangen können. Die throw-Anweisung beinhaltet im besten Fall nicht nur die ServletException, sondern zusätzlich eine IO-Exception.
Um die Anfragen spezifisch bearbeiten zu können, werden in Kapitel 5 einige spezielle-re Methoden vorgestellt, welche in der Methode `service` enthalten sind. Diese werden dann in den meisten Fällen anstelle von `service` überschrieben.

4 Programmierbeispiel

Das folgende Beispiel soll die Struktur und Aufgaben eines Servlets verdeutlichen (die Durchnummerierung der Zeilen dient der Übersicht in anschließenden Diskussion):

```
1:  import javax.servlet.*;
2:  import javax.servlet.http.*;
3:  import java.io.*;
4:  import java.util.*;

5:  public class SampleServlet extends HttpServlet {
7:      int numRequests = 0;
8:      String lastRequest = (new Date()).toString();
9:      Properties prop = null;

10:    public void init() throws ServletException {
12:        prop = newProperties();
13:        try {
15:            File file = new Fi-
                   le("SampleServlet.properties");

16:            if (file.exists()) {
18:                FileInputStream fileIn = new
                       FileInputStream(file);
19:                prop.load(fileIn);
20:                numRequests = Integer.parseInt(prop
                       .getProperty("RequestCount", "0"));
21:                lastRequest = prop.getProperty
                       ("LastRequest",
                       (new Date()).toString());
22:            } else {}
26:        } catch (Exception e) {}
30:    }
```

```
31:     public void service(HttpServletRequest request,
                HttpServletResponse response)
32:             throws ServletException, IOException {
34:         response.setContentType("text/html");
35:         PrintWriter out = response.getWriter();

36:         out.println("<HTML>");
37:         out.println("<HEAD><TITLE>Sample
                 Servlet</TITLE></HEAD>");
38:         out.println("<BODY>");
39:         out.println("<H1>Hello World!</H1>");
40:         out.println("<P>This servlet As been requested "
                + numRequests++ + " time(s).");
41:         out.println("<BR>Last requested at " +
                lastRequest + ".");
42:         out.println("</BODY></HTML>");

43:         lastRequest = (new Date()).toString();
44:         out.close();
45:     }

46:     public Sring getServletInfo() {
48:         return "Sample Servlet version 1.0";
49:     }

50:     public void destroy() {
52:         try {
54:             File file = new
                     File("SampleServlet.properties");
55:             FileOutputStream fileOut = new
                     FileOutputStream(file);
56:             prop.put("RequestCount",
                     Integer.toString(numRequests));
57:             prop.put("LastRequest", lastRequest);
58:             prop.store(fileOut, "SampleServlet Storage
                    File");
59:         } catch (Exception e) {}
63:     }
64:}
```

[Code 2: "Hello World"]

Die Zeilen 1 bis 4 importieren die benötigten Klassen und Interfaces. In den meisten Servlets werden auch Klassen aus dem Paket `javax.servlet` benötigt. Zur Vereinfachung wurde hier auf die Angabe der einzelnen Klassen verzichtet und alle Klassen durch den `.*` -Aufruf importiert.

Die Initialisierung benötigter Variablen und Objekte finden in den Zeilen 7 bis 9 statt. Die Variable `numRequests` gibt die Anzahl der eingegangenen Anfragen an, `lastRequest` speichert das Datum der letzten Anfrage als String.
Nach dem Aufruf der `service`-Methode in Zeile 31 wird zunächst in Zeile 34 der Typ des Outputs festgelegt ("text/html"). Die Methode `getWriter` in Zeile 35 wird benutzt, um die Ausgabe in Form eines Textes als Antwort zurückzugeben. Die Alternative für binäre Daten wäre die Methode `getOutputStream` gewesen. In den Zeilen 36 bis 42 wird die Antwort in Form von HTML geliefert. Die Ausgabe beinhaltet die Ausdrücke "Hello World!", "This servlet has been requested x times." und "Last requested at x.", wobei für x die entsprechende Anzahl beziehnungsweise das entsprechende Datum eingesetzt wird. Durch den Gebrauch von `numrequests++` (Zeile 40) wird die Anzahl der Requests direkt um eins erhöht. Das Datum des letzten Requests wird in Zeile 43 gesetzt. Als letzte Operation der `service`-Methode wird der Output-Stream geschlossen. Das sollte immer nach der letzten Ausgabe geschehen.

Die `getServletInfo`-Methode (Zeile 46 bis 49) liefert einen String zurück, der dem Server Informationen über das Servlet geben kann. Hier gibt `getServletInfo()` Name und Version des Servlets an (vgl. [call00]).

5 Java Servlet API - Die wichtigsten Komponenten

5.1 HttpServlet class

Die Klasse `HttpServlet` ist Bestandteil des Paketes `javax.servlet.http`. Sie erweitert die Klasse GenericServlet des Paketes `javax.servlet`. `HttpServlet` sollte die Oberklasse jedes Servlets sein.

Die Methode `service` wurde bereits in Kapitel 3.3 beschrieben. Sie behandelt jede Art von Requests im Gegensatz zu den sogenannten "`doXxx`"-Methoden. Diese behandeln nur eine Art von Anfrage. Bei spezialisierteren Servlets (z.Bsp. HTTP) sollten sie anstelle der `service`-Methode überschrieben werden. Das Überschreiben der `service`-Methode ist in diesem Fall sehr unüblich. Exemplarisch werden einige von der `doXxx`-Methoden jetzt vorgestellt:

- `void doGet (HttpServlet Request request, HttpServletResponse response)`
- `void doPost (HttpServletRequest request, HttpServletResponse response)`
- `void doPut (HttpServletRequest request, HtttpServletResponse response)`

...

Die Methode `doGet` wird bei einer GET-Anfrage aufgerufen. Der GET-Request ist der am häufigsten benutzte. Sollte `doGet()` (oder `service()`) nicht überschrieben worden sein, wird bei einer GET-Anfrage die Fehlermeldung HTTP 400 "Bad Request" ausgegeben. Diese Fehlermeldung verläuft bei allen Arten von Anfragen konform. Die Methode `doGet` wird als "protected" deklariert, weil es nur innerhalb der Servlet-Klasse aufgerufen wird. Die nachfolgenden doxxx-Methoden verhalten sich entsprechend. Deshalb wird hier nur noch auf die Art der Requests und eventuelle Besonderheiten eingegangen.

- Die Methode `doGet` kann unter anderem HEAD-Anfragen übernehmen. Dies erspart die Implementierung einer `doHead`-Methode.
- Wird `doHead` dennoch implementiert, liefert sie als Response ausschließlich Header zurück.
- POST-Anfragen sind die zweithäufigsten entgegengenommenen Anfragen. Um sie behandeln zu können, muß eine `doPost`-Methode implementiert werden.
- Um Daten des Clients hochladen zu können, werden PUT-Requests benutzt. Die Methode `doPut` behandelt diese.
- Die Anfrage eines Klienten, die Ressourcen in einem Request zu löschen ist ein DELETE-Request. Die entsprechende Methode heißt `doDelete`.

Die bis zu dieser Stelle genannten Methoden sind für ein Servlet notwendig. Neben diesen existiert noch eine Reihe weiterer Methoden in der `HttpServlet` Klasse, die entsprechende Anfragen verarbeiten können oder andere Funktionen übernehmen. Außer diesen existieren natürlich ebenfalls die Methoden der Klasse `GenericServlet`. Für eine vollständige Liste sei auf die Servlet API verwiesen. (vgl. [java02] und [call00])

5.2 HttpServletRequest Interface

Das `HttpServletRequest` Interface ist ein Subinterface des `ServletRequest`-Interfaces und Bestandteil des Paketes `javax.servlet.http`. Wenn ein Objekt das `HttpServletRequest`-Interface implementiert, wird es an die `doXxx`-Methoden weitergeleitet. Die Funktion des `HttpServletRequest`-Interfaces wird von den folgenden Methoden vorgestellt:

- `public Cookie[] getCookies()`

liefert eine Menge von Cookies zurück. Durch ein Cookie ist es dem Server möglich, ein Datenpaket zum Client zu senden, welches in jeder weiteren Anfrage dieses Clients verfügbar ist.

- `public String getMethod()`

liefert die Methode, die eine Anfrage benutzt. (Wie zum Beisiel POST, PUT, ...) Normalerweise besteht keine Notwendigkeit, `getMethod()` zu benutzen.

- `public String getAuthType()`

liefert den ein Authentifikationsschema zurück. Wird eine geschützte Datei angefragt, erzeugt der Server den Fehler HTTP 401 "Not Authorized". Wenn zum Beipiel nach einem Benutzernamen gefragt wird, wird in der Header der neuen Anfrage diesen beinhalten. Die Methode kann den Benutzernamen dann auf Korrektheit prüfen. Gegebenenfalls wird erneut der Fehler 401 erzeugt.

- `public String getRemoteUser()`

liefert den Benutzernamen nach einer Abfrage zurück zum Server. Wird kein Benutzername eingegeben, wird Null zurückgegeben.

Neben den 4 aufgelisteten Methoden existieren noch die `getXxxHeaderXxx`-Methoden. Sie arbeiten auf unterschiedliche Weise mit den entsprechenden Headern. Dazu gehört `getDateHeader()` und `getIntHeader()` sowie `getHeader()`, `getHeaders()` und `getHeaderNames()`.
Für die Methoden, die mit der URL arbeiten, wird im Folgenden nur ihre Rückgabe in Bezug auf ein Beipiel angegeben. Das jeweilge Beispiel ist dabei kursiv, die Rückgabe fett gedruckt.

`public String getPathInfo()`
http://localhost:8080/servlet/GetPathServlet/html/public?id=1234
/html/public

`public String getPathTranslated()`
http://localhost:8080/servlet/GetPathServlet/html/public?id=1234
C:\jakarta-tomcat\webpages\html\public

`public String getQueryString()`
http://localhost:8080/servlet/GetQueryServlet?name=Tyler&age=6
name=Tyler&age=6

`public String getRequestURI()`
http://localhost:8080/servlet/GetURIServlet/html/public?name=Tyler&age=6
http://localhost:8080/servlet/GetURIServlet/html/pulic

`public String getServletPath()`
http://localhost:8080/servlet/GetPathServlet/html/public?id=1234
/servlet/GetPathServlet

(Vgl. [call00] und [java02])

5.2 HttpServletResponse Interface

Das `HttpServletResponse`-Interface erbt vom `ServletResponse`-Interface. Es ist, wie auch das `HttpServletRequest`-Interface in dem Paket `javax.servlet.http` enthalten. Analog zum Verhalten des `Request`-Objektes wird

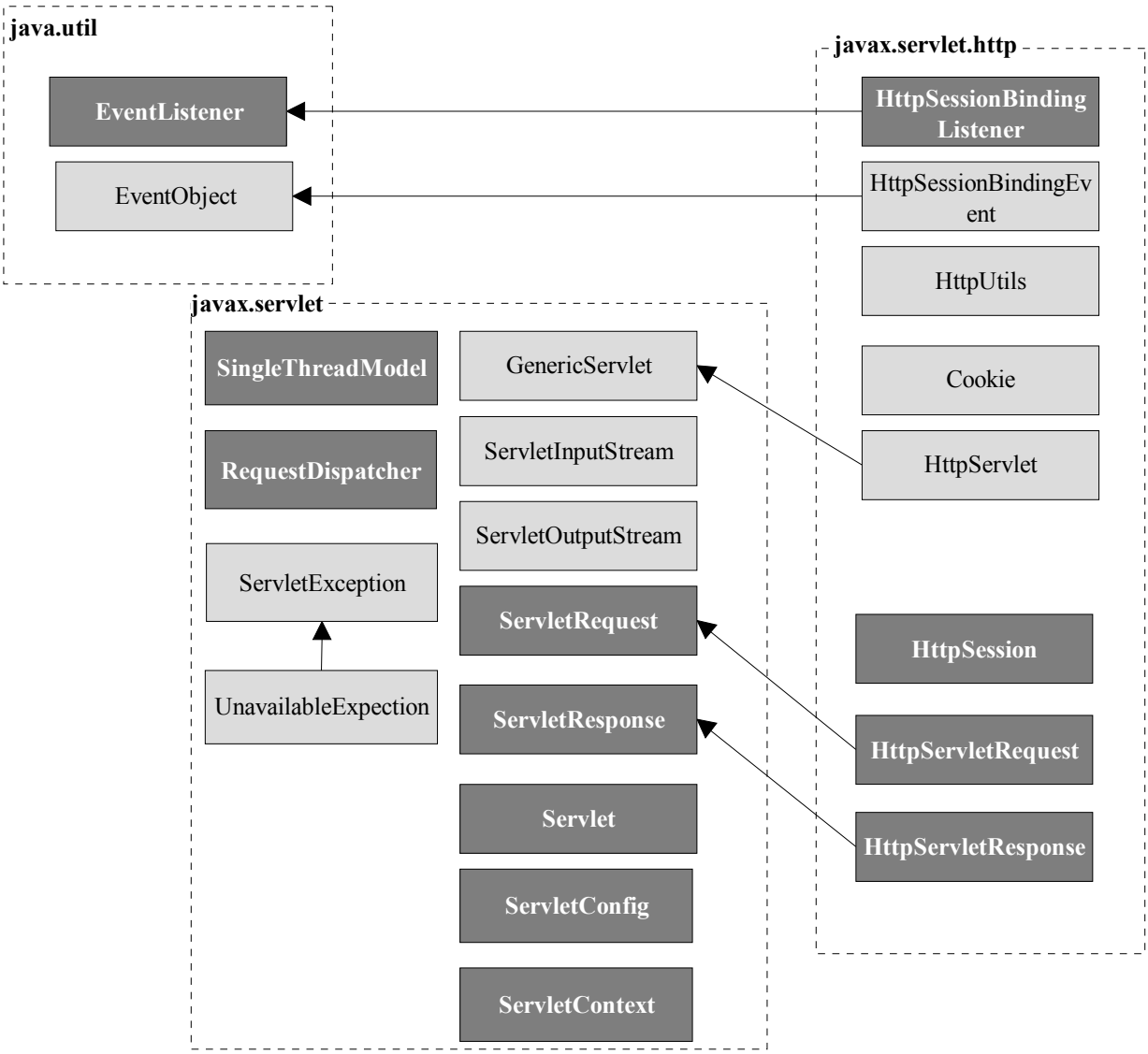
für die Antwort ein Objekt, welches `HttpServletResponse` implementiert, von der Methode `service` des Servlets benutzt, um auf den Anfrage zu reagieren. Kooperierend mit den Methoden des `HttpServletRequest`-Interfaces sind zum Beipiel folgende Methoden:

```
public void setHeader(String name, String value) bzw.
public void addHeader(String name, String value)
```

Mit `setHeader` werden Header mit dem übergebenen Namen überschrieben. Im Gegensatz dazu wird mit `addHeader` ein neuer Header hinzugefügt, ohne die übrigen Header zu überschreiben. So kann ein Header mehrere Werte enthalten.
Die Methoden `setDateHeader` und `addDateHeader` sowie `setIntHeader` und `addIntHeader` verfahren analog. Die Parameter sind allerdings bei `XxxDateHeader()` `date` vom Typ `long` und `bei XxxIntHeader() value` vom Typ int anstelle der Zeichenkette in `XxxHeader()`. Neben diesen Header-Methoden existiert die Methode `containsHeader`: `public boolean containsHeader(String name)`. Diese findet heraus, ob ein spezieller Header Inhalt der Antwort ist. Der Rückgabewert ist entsprechend `true` oder `false`. Eine weitere analoge Methode ist `addCookies`, die es erlaubt, ein Cookie an eine Antwort anzuhängen. (siehe auch `getCookies()` aus Kapitel 5.2)
Durch die Methode `sendError` ist es möglich, eine Fehlermeldung zu senden. Diese Fehlermeldung ist abhängig von einem sogenannten Statuswert, wie SC_CONFLICT. Dieser Statuswert beispielsweise drückt aus, daß der Zugang zu den erbetenen Daten aufgrund eines Konfliktes verweigert wurde. Die Methode erlaubt zwei verschiedene Formen:
`public void sendError(int sc)` oder
`public void sendError (int sc, String message)`
wobei die erste als Parameter nur die Konstante des Statuswertes akzeptiert, während die zweite Methode auch eine kurze zugehörige Erklärung zulässt.
Ähnlich wie `sendError()` kann auch `setStatus()` den Statuswert an die Antwort anhängen, nur das dieser für gewöhnlich natürlich keine Fehlermeldung nach sich zieht.
`public void sendRedirect(String location)` sendet eine Nachricht an den Klienten in Bezug auf die URL, zu der der Client weitergeleitet wird (vgl. [call00] und [java02]).

5.3 Übersicht

Die Abbildung 4 zeigt die Zusammenhänge der einzelnen Klassen, Interfaces und Paketen.
Die gestrichelten Kästen sind dabei die einzelnen Pakete, die hellgrau unterlegten Kästen sind Klassen und die dunkelgrau unterlegten Kästen Interfaces. Die Pfeile kennzeichnen die Vererbung.



[Abb. 4: Hierarchie]

6 Zusammenfassung

In Kapitel 1 wurde über die Motivation berichtet, Servlets zu entwickeln und benutzen. Dazu gehören unter anderem die vielen Vorteile von Servlets, die es erlauben, dynamisch Daten zwischen dem Server und dem Klienten auszutauschen. Durch den Gebrauch von Java sind Servlets plattform-, server- und protokollunabhängig im Gegensatz zum Beispiel zu NSAPI und ISAPI und einfach und sicher zu entwickeln. Außerdem sind Servlets multithreadfähig, was die Leistung im Vergleich zu CGI oder FastCGI erhöht.
Die Leistung von Servlets liegt darin, daß Server und Klient miteinander kommunizieren können. Servlets nehmen Anfragen des Klienten an und liefern die entsprechende

Antwort zurück. Weiterhin werden in Kapitel 2 die unterschiedlichen Arten von Containern erklärt. Container stellen eine Laufzeitumgebung für die Komponenten da. Hierzu gehörten "stand-alone servlet container", "in-process servlet container" und "out-of-process servlet container".
Im dritten Kapitel wird der Lebenszyklus eines Servlets anhand einer Grafik erklärt. Dieser besteht aus 8 Schritten. Die ersten drei Schritte betreffen das Aufrufen, Laden und Initialisieren des Servlets. Soll das Servlet arbeiten, wird durch `service()` eine eingehende Anfrage bearbeitet. Falls das Servlet zerstört werden soll, wird `destroy()` aufgerufen. Es ist möglich, das Servlet neu zu laden, um Änderungen ohne einen Neustart des Servers zu übernehmen. Anhand von Code 1 werden die drei wichtigsten Methoden erklärt.
Das vierte Kapitel besteht aus einem Programmierbeispiel. Code 2 liefert ein Servlet, welches den Satz "Hello World!" ausgibt, so wie die Anzahl der Anfragen und das Datum der letzten Anfrage.
Als letztes wird ein kurzer Einblick in die Servlet API gegeben. In der Klasse `HttpServlet` wurden vor allem die Methoden `doGet`, `doPost` und `doPut` erklärt, welche die entsprechende Anfrage (GET, POST oder PUT) verarbeiten. Die wichtigsten Methoden des `HttpServletRequest`-Interfaces und des `HttpServletResponse`-Interfaces wurden mit den entsprechenden Ausgaben ebenfalls in Kapitel 5 behandelt.

7 Quellenangaben

[call00] Dustin R. Callaway, Addison Wesley. *Inside Servlets*
[raas01] Jörg Raasch . *Konzeptbericht zum Thema: TOMCAT* (2001);
http://users.informatik.fh-hamburg.de/~raasch/SEP-Webseiten/Berichte/tomcat.html

[java02*] Products & API*; http://java.sun.com/products
[serv01] Servlets - Eine Einführung, 2001;
http://www.independent-web.de/advanced/java/servlets_voraussetzungen.html

[koel99] Peter Köller. Servlets und JavaServer-Pages, 1999/2000;
http://www.metaprojekt.de/Downloads/servlets.pdf
[jaka02] http://jakarta.apache.org