

10 World-Wide Web

10.1 HTTP	2
10.2 URLs	10
10.3 Dynamische Ressourcen	13
10.3.1 CGI	17
10.3.2 Java Servlets	23
10.3.3 Formularverarbeitung (wird fortgesetzt)	27
Anhang 1: HTTP-Kopfzeilen	31
Anhang 2: HTTP-Statuscodes	42
Anhang 3: CGI-Umgebungsvariable	49
Zusammenfassung	52

Antworten auf die Frage „Was ist das WWW?“:

- ① eine Menge weltweit installierter **Server mit Port 80**, deren Dienst im Herunterladen von Dateien besteht; Anfragen und Antworten sind in der Regel **textuell** und befolgen des Protokoll **HTTP** (oder **HTTPS**).
- ② eine Menge weltweit vorhandener und öffentlich verfügbarer Dateien, meist **HTML**-codiert, die Verweise auf andere Dateien enthalten können.
- ③ eine **Plattform** für verteilte Anwendungen mit Server als generischem **Backend** und Browser als generischem **Frontend** - evtl. mit Beteiligung verschiedener Sprachen.

10.1 HTTP

(hyper-text transfer protocol)

(Beachte: unglücklicher Name, weil nicht nur
Hypertext (HTML) übertragen wird !)

- Klient - z.B. Browser - öffnet TCP-Verbindung zum Server und schickt **Anfragetext** an den Server mit dem Wunsch nach einem **Dokument** (etwa Inhalt einer Datei, „Webseite“)
- allgemeiner: Aktivierung einer **Ressource** (*resource*).
- Server schickt **Antworttext** mit beigefügtem Dokument und schließt die TCP-Verbindung (!).

Klient

```
GET /index.html HTTP/1.0  
User-Agent: Mozilla/5.0 .....  
.....
```

(Leerzeile beendet die Anfrage!)

Server

```
HTTP/1.1 200 OK  
Date: Thu, 30 Dec 2010 15:23:09 GMT  
Server: Apache/2.2.8 (Unix) .....  
(Leerzeile beendet die Antwort!)  
<!DOCTYPE HTML PUBLIC "-//W3C//DTD ... >  
<HTML>  
<HEAD>  
<TITLE>.....  
.....
```

\$ nc www.inf.fu-berlin.de 80

GET /lehre/WS12/alp5/literatur.html HTTP/1.0

(Leerzeile beendet die Anfrage!)

HTTP/1.1 200 OK

Date: Tue, 08 Jan 2013 16:36:27 GMT

Server: Apache

Last-Modified: Sun, 14 Oct 2012 15:10:40 GMT

ETag: "2b211fd-1115-4cc0652ed5c00"

Accept-Ranges: bytes

Content-Length: 4373

Vary: Accept-Encoding

Content-Type: text/html

Via: 1.0 www.inf.fu-berlin.de

Connection: close

(Leerzeile beendet die Antwort!)

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/standards/official.dtd">

<html><head>

<meta content="text/html; charset=ISO-8859-1" http-equiv="content-type">

<h4>FU Berlin, WS 2012/13</h4>

.....

➤ **Anfrage-Syntax:**

```
method resource HTTP/x.y  
headers  
[ body ]
```

- **method:** **GET**, **PUT**, **POST**, **HEAD** und andere Methoden
- **resource** identifiziert Dokument - allgemeiner: **Ressource**
 - absoluter Pfad im Server-Verzeichnisbaum
 - (vollqualifizierte URL bei Anfrage an *Proxy Server*)
 - (*****, **authority** bei bestimmten Methoden)
- **x.y** ist die Version, **1.0** oder **1.1**
- **headers:** einige Kopfzeilen (siehe Anhang 1)
- **body** (optional): mitzuschickende Daten

➤ **POST**

- Übertragung von Daten, die der Benutzer in ein *HTML-Formular* eingetragen hat, zum Server
- **POST /some.cgi HTTP/1.0**
..... <gefolgt von Daten>
- wird beantwortet mit Code, Kopfzeilen, eventuell Inhalt

➤ (**PUT** - i.d.R. nicht öffentlich verfügbar)

- Hochladen einer Datei zum Server
- **PUT /index.html HTTP/1.0**
..... <gefolgt von Daten>
- wird beantwortet mit Code und Kopfzeilen

➤ (weitere ...)

➤ Antwort-Syntax:

HTTP/x.y code phrase
headers
[body]

- x.y ist die Protokoll-Version, 1.0 oder 1.1
- code: Statuscode numerisch (*status code*), z.B. 200 (siehe Anhang 2)
- phrase: Statuscode verbal (*reason phrase*), z.B. OK
- headers: einige Kopfzeilen (siehe Anhang 1)
- body (optional): mitgeschickte Daten

! netcat hilft bei der Protokoll-Analyse, z.B. wenn man selbst entwickelte Server/Browser testen will !

- *Anfrage eines Browsers analysieren:*
eigenen „Server“ bereithalten, der eine erhaltene Anfrage wörtlich ausdrückt, z.B. so (mit `-l` für „listen“):
`nc -l 56789` und dann im Browser `http://localhost:56789`
- *Antwort eines Servers analysieren:*
eigenen „Browser“ benutzen, der eine erhaltene Antwort wörtlich ausdrückt, z.B. so:
`nc localhost 80` und dann Anfrage eingeben (S. 5)
- (Nebenbei bemerkt: `nc -l 12345 & nc localhost 12345` echot die Eingabezeilen !)

10.2 URLs

URI: Uniform Resource Identifier

„..... is a compact string of characters for identifying an abstract or physical resource“ [RFC 2396]

URN: Uniform Resource Name is a URI

„... persistent, location-independent, resource identifier, designed to make it easy to map other namespaces into URN-space“ [RFC 2141]

URL: Uniform Resource Locator is a URI

„... is a compact string representation for a resource available via the Internet.“ [RFC 1738]

URI-Syntax:

absoluteURI = *scheme* : *scheme-specific-part*
scheme-specific-part =

Beispiel URN:

urn:isbn:3-89864-421-9

Beispiele URL:

ftp://ftp.inf.fu-berlin.de/pub/readme
http://www.inf.fu-berlin.de:80
https://fahrrkarten.bahn.de/privatkunde/...
file:/Users/lohr/tmp/abc

Weitere Beispiele:

mailto:lohr@inf.fu-berlin.de
.....

Syntax der URLs für HTTP/HTTPS:

scheme :// host : port / path ? querystring
(Protokoll Rechner Port Weg weitere Daten)
`http://blue.org:8000/res/x?query=berlin`

URLs in Java:

Klasse `java.net.URL` für URL-Objekte

Klasse `java.netURLConnection` kümmert sich beim Kontakt mit einem Server um die protokollspezifischen Daten, so dass der Programmierer nur noch mit den eigentlichen Nutzdaten zu tun hat.

10.3 Dynamische Ressourcen

Web Server wird zum **generischen Server** verallgemeinert:

Dynamische Ressourcen anstelle **statischer Dokumente**, d.h.

- URL repräsentiert Code-Aufruf samt Argumenten,
- Code liefert ein generiertes Dokument als Ergebnis.

<http://www.google.de/search?q=berlin+mauer&hl=de&...>



Ressource „Programm **search**“



Argumente **q, hl, ...**
(max. 255 Zeichen)

Konvention (!) für die Verwendung von Argumenten:

Nach dem Trennzeichen `?` folgt ein **query string** :

- keine Leerzeichen;
- Argumente der Form `name = value`,
- getrennt durch `&` ;
- URL-codiert, d.h. ggfls. Darstellung syntaktischer Zeichen durch andere, z.B. `+`, `%26` (statt `&`), `%2B` (statt `+`), ...
(siehe `java.net.URLEncoder`, `java.net.URLDecoder`)

Der Web Server stellt den *query string* dem Code der dynamischen Ressource in geeigneter Weise zur Verfügung:

entweder über die Umgebungsvariable `$QUERY_STRING`
oder über die Standard-Eingabe (bei `POST`).

Einige Techniken (in historischer Reihenfolge):

- **CGI** (*Common Gateway Interface*): beliebiger Code.
Für die Ausführung erzeugt der Server einen Prozess.
- **Servlets**: Java-Code.
Für die Ausführung setzt der Server einen Thread ein.
- **PHP** (*PHP Hypertext Processor*): Skriptsprache,
eingebettet in HTML und von Server-Thread ausgeführt.
- **JSP** (*Java Server Pages*): Java-Code,
eingebettet in HTML und von Server-Thread ausgeführt.
- **ASP** (*Active Server Pages*, Microsoft): Skriptsprachen,
eingebettet in HTML und von Server-Thread ausgeführt.
- **ASP.NET** (Microsoft): C#-Code und HTML getrennt,
aber mit wechselseitigen Bezugnahmen.

Klassifikation dynamischer Ressourcen:

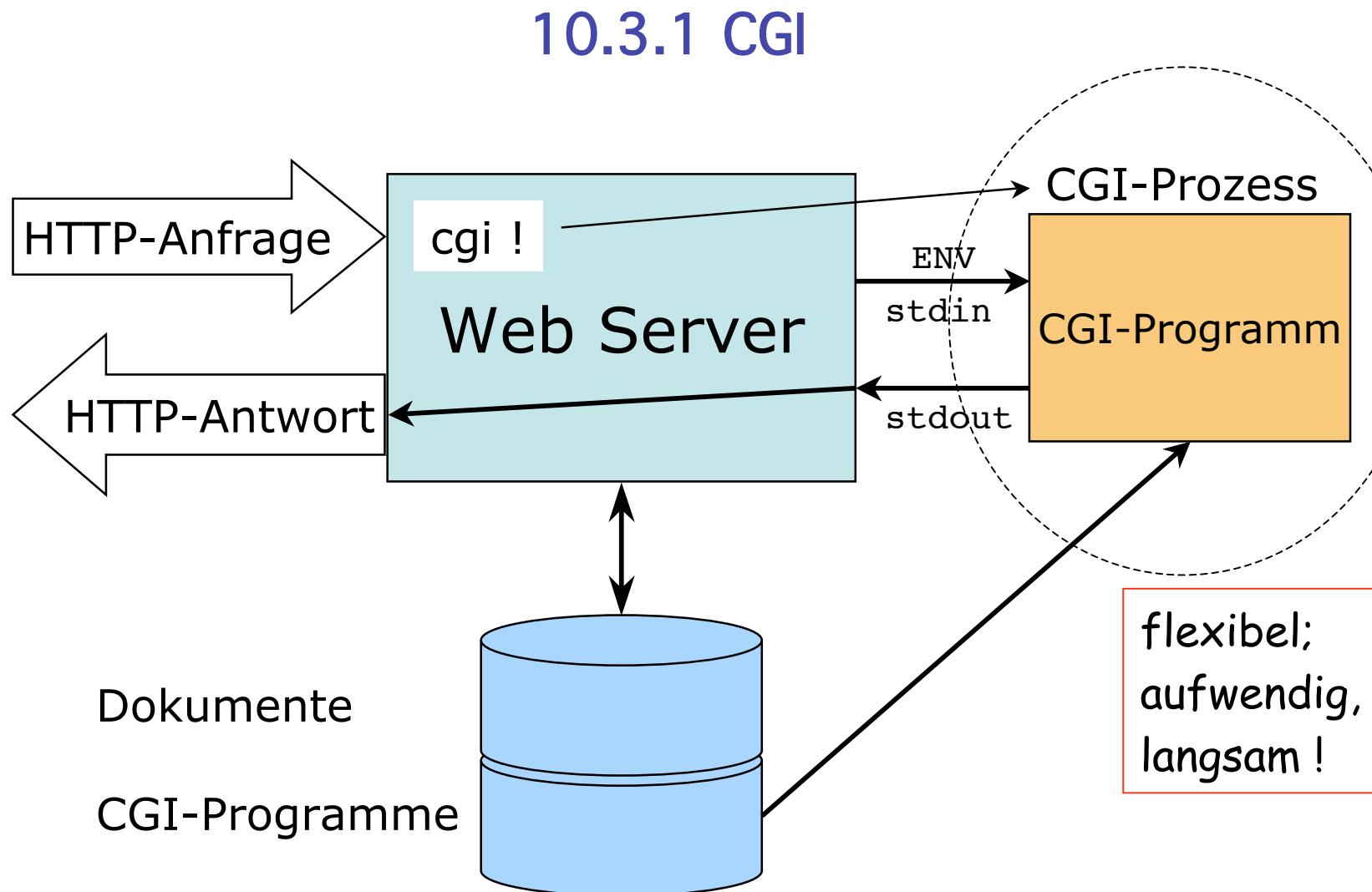
programmorientiert: Code erzeugt HTML-Text;
„HTML ist eingebettet in Code“. (CGI, Servlets)

dokumentorientiert: Server modifiziert vorliegenden HTML-Text
durch Ausführung dort enthaltener Code-Fragmente;
„Code ist eingebettet in HTML“. (PHP, JSP, ASP)

(Bemerkung: Server beherrscht u.U. nicht alle diese Techniken !)

Beachte:

Es handelt sich hier um elementare Server-Techniken.
Dem Entwickler einer Web-Anwendung unter Einsatz
eines **Web Framework** bleiben sie häufig verborgen.



Populär - wenngleich nicht notwendig - ist **Skript-Code**.

Damit dieser als Code und nicht als Daten interpretiert wird:
CGI-Programme entweder mit Dateierweiterung **.cgi** oder
in Verzeichnis **cgi-bin**, z.B.

`http://www.lob.de/cgi-bin/work/framesetneu`

`http://page.mi.fu-berlin.de/lohr/cgi-bin/date.rb`

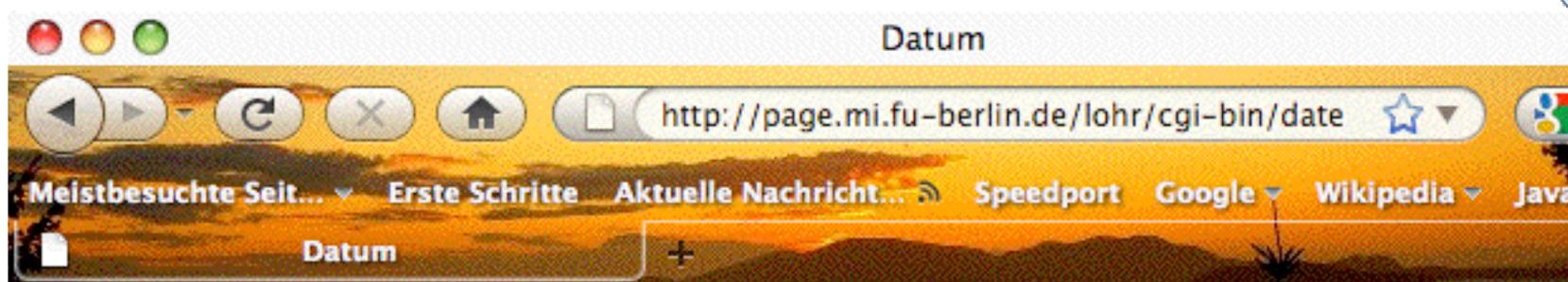
Beachte: execute-Recht an CGI-Datei erforderlich

Öffentliches CGI-Programm: mit Server-Rechten ausgeführt

Privates CGI-Programm: mit Benutzer-Rechten ausgeführt

www.mi.fu-berlin.de/w/Tec/ItServicesPersoenliche_Webseiten

Beispiel: 4 Alternativen für dieses Resultat:



Heutiges Datum: Thu Jan 10 21:35:10 2013

Fertig

```
#!/usr/bin/ruby
puts "Content-Type: text/html"      # kopfzeile
puts
puts "<html><head><title>Datum</title></head>"
puts "<body>Heutiges Datum: "+Time.now.ctime()
puts "</body></html>"
```

```
#!/bin/sh
java Date
```

(übersetztes
date.c)

```
#!/bin/sh
echo Content-Type: text/html
echo
echo "<html><head><title>Datum</title></head>"
echo "</head><body>"
echo Heutiges Datum:
date
echo "</body></html>"
```

(... weitere
Möglichkeiten)

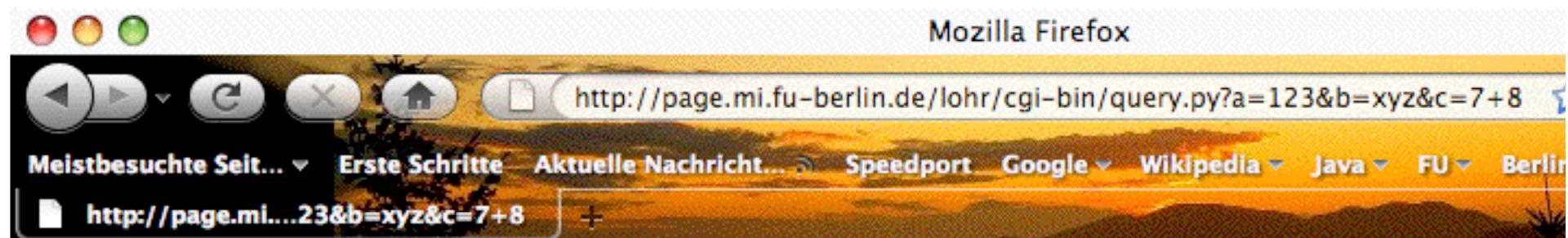
Verwendung von Argumenten

Das CGI-Programm erhält den *query string* (S. 14)
entweder über die Umgebungsvariable `$QUERY_STRING`
oder über die Standard-Eingabe (bei `POST`).

Manche Skriptsprachen kennen die Syntax des *query string* und erlauben den direkten Zugriff auf die benannten Argumente, z.B.

```
#!/usr/bin/python      (→ Effekt siehe S. 21)
import cgi
query = cgi.FieldStorage()
print "Content-Type: text/html"
print
for key in query:
    print '<br>', key, 'has value', query[key].value
```

Beachte: Alles ist textbasiert, keine Typen, keine Typsicherheit !



Fertig

(Python FieldStorage ist ein *HashMap*)

Laufzeitfehler des CGI-Programms führt zur Fehlermeldung

500 Internal Server Error

die vom Browser geeignet angezeigt wird.

Achtung: **Sicherheit** ist bedroht durch gezielt „falsche“
Benutzung durch Angreifer !

Die Art der Bedrohung hängt von der verwendeten
Programmiersprache ab (z.B. Pufferüberlauf bei
unsicheren Sprachen - siehe 6.5).

Beachte: CGI-Programm wird mit den Rechten
des Servers oder des Eigners ausgeführt
(vgl. *setuid*-Programme).

10.3.2 Java Servlets

Wenn der Web Server selber auf einer JVM läuft:
Java Servlets in leichtgewichtigen JVM Threads
statt CGI-Programme in schwergewichtigen Prozessen !

Unterschied zu CGI:

Ein/Ausgabe nicht explizit über Umgebungsvariable
bzw. Standard-Ein/Ausgabe, sondern über eine
vom *Servlet* zu implementierende Schnittstelle.

In J2EE diverse einschlägige Pakete: `javax.servlet`
`javax.servlet.http`
...

Beispiel S. 20/21:

das Python-Skript `query.py` wird durch die folgende Java-Klasse ersetzt, die im *Servlet Container* des Web Servers bereitgestellt wird. (Die Zuordnung zwischen Klassename und URL-Pfad hängt vom Server ab.)

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class QueryServlet extends HttpServlet {
    public void doGet(HttpServletRequest req,
                      HttpServletResponse res)
        throws ServletException, IOException {
        ....
```

```
.....
String a = req.getParameter("a");
String b = req.getParameter("b");
String c = req.getParameter("c");

res.setContentType("text/html");
PrintWriter out = res.getWriter();

out.println("<br>a has value " + a);
out.println("<br>b has value " + b);
out.println("<br>c has value " + c);
}

}
```

- Achtung: nur Java-basierte Web Server verfügen über einen Container für Servlets und unterstützen das Servlet-Konzept.
- Einheit der Installation im Container ist das *Web Application Archive*, kurz **WAR**.
- Unterstützung von Servlets:
 - Apache Tomcat - <http://tomcat.apache.org>
 - JBoss - <http://www.jboss.com>
 - Google App Engine - <https://developers.google.com/appengine/>
 - ... weitere
- Keine Unterstützung von Servlets:
 - Apache HTTP Server - <http://httpd.apache.org>
 - MS IIS - <http://www.iis.net>

10.3.3 Formularverarbeitung

Formular (*form*) ist ein HTML-Element.

Es stellt die einfachste Möglichkeit dar, in eine Webseite eine graphische Benutzerschnittstelle (GUI) einzubauen.

Der Benutzer kann in das Formular Informationen eintragen und dann das Formular „abschicken“: es erfolgt eine weitere Server-Anfrage - an eine dynamische Ressource; im *query string* bzw. aus der Standard-Eingabe erhält die dynamische Ressource die Formulardaten.

Ein Beispiel-Formular:

Welcome to the Gothic Club!

Check your eligibility for membership:

First name: Adam W. Last name: Smith

Age: 23

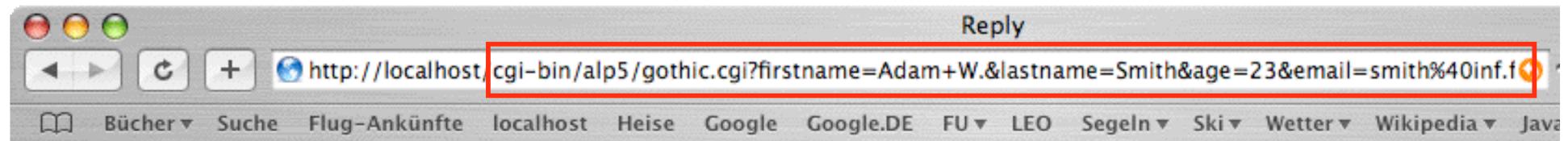
email: smith@inf.fu-berlin.de

Sex: male female

Send **Reset**

(oder Eingabe-Taste)

Effekt: erneute Anfrage an den Server, CGI-Programm berechnet neue Seite, diese wird ausgeliefert:



Sorry, Adam, you are not eligible.

Der *query string* hinter dem ? ist URL-codiert (S. 14) !

Der HTML-Text in der Datei gothic.html:

```
<!DOCTYPE html PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<HTML><TITLE>Welcome to the Gothic Club</TITLE>
<h2>Welcome to the Gothic Club!</h2>
Check your eligibility for membership:<br><br>
<FORM action="http://localhost/cgi-bin/alp5/gothic.cgi"
      method="GET">
    First name: <INPUT type="text" name="firstname" size=10>
    Last name: <INPUT type="text" name="lastname"><br>
    Age: <INPUT type="text" name="age" size=2><br>
    email: <INPUT type="text" name="email"><br>
    Sex: <INPUT type="radio" name="sex" value="Male"> male
          <INPUT type="radio" name="sex" value="Female"> female
    <br><br>
    <INPUT type="submit" value="Send"> <INPUT type="reset">
</FORM>
</HTML>
```

(Übung: geeignetes gothic.cgi schreiben!)

Anhang 1: Kopfzeilen

... für Anfragen und Antworten:

- **Date:** *datetime*

Datum/Zeit des Abschickens der Nachricht im
RFC-1123-Format, z.B. `Sat, 06 Dec 2008 10:07:18 GMT`

- **Connection:** *option*

Verbindung wird nach Ablauf von Frage/Antwort

- gelöst mit *option close*
- nicht gelöst mit *option keep-alive*
-

- **Via:** *protocolVersion host ...*

Weg der Nachricht, z.B.

`Via: 1.0 fred, 1.1 nowhere.com (Apache/1.1)`

- **Cache-Control:** directive
Steuert das *Caching* von Anfragen und Antworten
 - **no-cache:** Antwort darf nicht zur Beantwortung anderer Anfragen genutzt werden
 - **no-store:** Antwort- oder Anfragemitteilungen dürfen nicht gespeichert werden
 - **weitere:** **max-age**, **max-stale**, **min-fresh**, **no-transform**, **only-if-cached**, **public**, **private**, **must-revalidate**, **proxy-revalidate**, **s-maxage**
- **Pragma: no-cache**
entspricht **Cache-Control: no-cache**

Kopfzeilen für Anfragen:

- **Host: name**
Name des Servers. *Obligatorisch in HTTP 1.1*
- **If-Modified-Since: datetime**
Forderung nach hinreichend aktuellem Dokument; Antwort:
 - Code **200 OK** und Inhalt schicken,
 - bzw. Code **304 Not Modified** und Inhalt *nicht* schicken.
- **If-Unmodified-Since: datetime**
Forderung nach hinreichend altem Dokument; Antwort:
 - Code **200 OK** und Inhalt schicken,
 - bzw. Code **412 Precondition Failed** und Inhalt *nicht* schicken.

- **Range: range**

Nur ein *Fragment* des Dokuments wird angefordert; Antwort:
Code 206 Partial Content und Inhalt schicken. Beispiel:

```
$ nc www.inf.fu-berlin.de 80
GET /lehre/WS08/alp5/index.html HTTP/1.0
Range: bytes=0-75
```

```
HTTP/1.1 206 Partial Content
Date: Sat, 06 Dec 2008 15:16:27 GMT
.......
```

- **From: mailaddress**

Absender-Identifikation - typischerweise *nicht* angegeben und -
wenn angegeben - nicht benutzt.

- **User-Agent:** *product / version (comments)*

Identifikation des Klientenprogramms, i.d.R. ein Browser, z.B.

Mozilla/5.0 (Macintosh; U; PPC Mac OS X 10_4_11;....)

- Browser können sich „maskieren“, indem sie sich hier *nicht korrekt identifizieren*.
- Der Server kann - wenn er will - je nach Browser unterschiedlichen Inhalt ausgeben (Smartphones, Tablets, ...!)

- **Referer:** *URL*

Die URL der Webseite, auf der ein *Link* zu dieser Anfrage führte.

! Datenschützer lieben das gar nicht !

- **Authorization:** *authdata*
Autorsierungsnachweis, z.B.
`Authorization: username="Mufasa",
realm="testrealm@host.com",
response="6629fae49393a05397450978507c4ef1"`
- **Proxy-Authorization:** *authdata*
Autorsierungsnachweis für Proxy
- **Max-Forwards:** *number*
Maximale Anzahl der Weiterleitungen einer OPTIONS- oder TRACE-Anfrage über Gateways oder Proxies
- **Expect:** *token*
Klient erwartet bestimmte Eigenschaften des Server/Proxy.
Im negativen Fall Fehlercode 417.

Kopfzeilen für Antworten:

- **Date:**
Zeitpunkt des Abschickens der Ressource
- **Server:** *productDescription*
Server-Produkt, z.B. **Server:** Apache/1.3.41 (Darwin)
- **Last-Modified:** *date*
Zeitpunkt der letzten Änderung der Ressource
- **Accept-Ranges:** *token*
Möglichkeit der Teilübertragung, z.B.
Accept-Ranges: bytes oder **Accept-Ranges:** none
- **Retry-After:** *date*
Bei 503: Zeitpunkt zur Wiederholung der Anfrage, z.B.
Retry-After: Fri, 31 Dec 1999 23:59:59 GMT
Retry-After: 120 (Sekunden)

- **Age:** *seconds*
Geschätztes Alter der Ressource
- **Location:** *URI*
Bei 201: Adresse der neu geschaffenen Ressource
Bei 3xx: Adresse für Umlenkung
- **Expires:** *date*
Kann nach dem angegebenem Datum aus dem *Cache* gelöscht werden
- **WWW-Authenticate:** *credentials*
Klient muss sich gegenüber Server ausweisen, sonst Fehler 401
- **Proxy-Authenticate:** *credentials*
Klient muss sich gegenüber Proxy ausweisen, sonst Fehler 407
- **Connection:** *closeOrOther*

Inhaltsbeschreibende Kopfzeilen:

- **MIME-Version:** *version*
- **Content-Language:** *language*
Sprache des Inhalts: *de, en, en-US, ...*
- **Content-Encoding:** *encoding*
Kompression des Inhalts: *deflate, gzip, ...*
- **Content-Type:** *mediatype*
Medientyp des Inhalts: *text/plain, image/gif, ...*
(MIME media type, www.iana.org/assignments/media-types/)

- **Content-Length:** *length*
Länge des Inhalts in Byte
- **Content-Range:** *range*
Beschreibung des Ausschnitts bei Teilanforderung
- **Content-Location:** *URI*
Adresse der Ressource
- **Content-MD5:** *MD5checksum*
Hashwert (*message digest*) für Integritätsprüfung des Inhalts

Anhang 2: Statuscodes (Auszug)

- *Informational 1xx: experimentell*

➤ *Successful 2xx: Erfolgreiche Ausführung*

- 200 OK
GET, HEAD, POST, TRACE erfolgreich, Antwort anbei
- 201 Created
Erfolgreiches PUT oder POST
- 202 Accepted
Für spätere Ausführung vermerkt
- 203 Non-Authoritative Information
Metainformationen in Kopfzeilen stammen von Dritten
- 204 No Content
Anfrage verarbeitet, kein Antwortinhalt notwendig
- 205 Reset Content
Anfrage verarbeitet, Ansicht erneuert
- 206 Partial Content
GET mit Teilanforderung erfolgreich, Teilantwort anbei

➤ *Redirection 3xx: neue, modifizierte Anfrage erforderlich*

- 300 **Multiple Choices**

Verschiedene Versionen erhältlich, **Accept**-Kopfzeile nicht eindeutig

- 301 **Moved Permanently**

Verschoben (**Location**-Kopfzeile gibt Auskunft - s.u.)

- 302 **Found Moved Temporarily**

Verschoben (**Location**-Kopfzeile gibt Auskunft - s.u.)

- 303 **See Other**

Andere URI benutzen (**Location**-Kopfzeile gibt Auskunft - s.u.)

- 304 **Not Modified**

Bei GET mit **If-Modified-Since**-Kopfzeile

- 305 **Use Proxy**

Muss über Proxy angesprochen werden (**Location**-Kopfzeile gibt Auskunft - s.u.)

- 307 **Temporary Redirect**

Umleitung bei **GET, HEAD**

➤ *Client Error 4xx: Fehlerhafte Anfrage des Klienten*

- 400 **Bad Request**
Falsche Anfragesyntax
- 401 **Unauthorized**
Passwort erforderlich ([WWW-Authenticate-Kopfzeile](#) - s.u.)
- 403 **Forbidden**
Ohne Angabe von Gründen verweigert
- 404 **Not Found**
Ressource nicht auffindbar
- 405 **Method Not Allowed**
Methode für die Ressource nicht zugelassen
- 406 **Not Acceptable**
Ressource passt nicht zu [Accept](#)-Kopfzeilen
- 407 **Proxy Authentication Required**
Passwort für Proxy erforderlich ([Proxy-Authenticate-Kopfzeile](#))
- 408 **Request Timeout**
Zeitüberschreitung bei Übermittlung der Anfrage

- 409 Conflict

Methode steht in Konflikt mit Zustand des Servers, Client kann Konflikt aufheben

- 410 Gone

Ressource permanent nicht auffindbar

- 411 Length Required

Content-Length-Kopfzeile erforderlich

- 412 Precondition Failed

Bedingungen der Anfrage (in Kopfzeilen) unerfüllbar

- 413 Request Entity Too Large

Anfrage zu groß

- 414 Request URI Too Long

URI zu lang

- 415 Unsupported Media Type

Unbekanntes Inhaltsformat

- 416 Requested Range Not Satisfiable

Teilanforderung falsch beschrieben

- 417 Expectation Failed

Expect-Kopfzeile unerfüllbar

➤ *Server Error 5xx: Fehlfunktion beim Server*

- 500 Internal Server Error
- 501 Not Implemented

Angeforderte Methode wird nicht unterstützt

- 502 Bad Gateway

Weiterer benutzerter Server nicht erreichbar

- 503 Service Unavailable

Server kann Dienst gerade nicht erbringen (vgl. Kopfzeile **Retry-After** - s.u.)

- 504 Gateway Timeout

Weiterer benutzerter Server antwortet nicht rechtzeitig

- 505 HTTP Version Not Supported

Unbekannte HTTP-Version

Anhang 3: Umgebungsvariable eines CGI-Programms

- SERVER_NAME
Der Name des Servers, wie er in der URL des CGI-Programms auftritt.
- SERVER_PORT
Die Nummer des Ports, über den die Anfrage geschickt wurde.
- SERVER_PROTOCOL
Das Protokoll, mit dem die Anfrage geschickt wurde, als Zeichenkette der Form Protokoll/Version. Z.B. HTTP/1.1
- SERVER_SOFTWARE
Server, der das CGI-Programm startet, als metaName/Version. Beispiel:
NCSA/1.4.2.
- GATEWAY_INTERFACE
Version der CGI-Spezifikation, der der Server folgt, als CGI/Version. Z.B.:
CGI/1.1.
- REQUEST_METHOD
Methode, die bei der Anfrage verwendet wurde: GET oder POST (s.u.)
- SCRIPT_URI
Vollständiger URI des CGI-Programms
- SCRIPT_NAME
Der Name des CGI-Programms in der URI, z.B. [/cgi-bin/env.cgi](#)

- QUERY_STRING
Die Eingabe/Anfrage an das CGI-Programm
- REMOTE_HOST
Der Name des Rechners, von dem die Anfrage kam
- REMOTE_ADDR
Die Internetadresse des Rechners, von dem die Anfrage kam
- REMOTE_USER
Der Benutzername, für den das Passwort eingegeben wurde
- REMOTE_IDENT
Benutzerkennung des anfordernden Benutzers, falls ermittelbar
- AUTH_TYPE
Name des Verfahrens, mit dem das Passwort kodiert ist
- CONTENT_TYPE
Bei Verwendung der POST-Methode steht hier der Medientyp des Inhalts, der an der Standardeingabe gelesen werden kann
- CONTENT_LENGTH
Die Länge des Inhalts bei der POST-Methode
- HTTP_ACCEPT
Die Medienarten, die der Browser akzeptieren will

Zusammenfassung

- Das Protokoll HTTP ist *textbasiert* und regelt die Anfragen/Antworten zwischen Web Server und Klient.
- URIs/URLs identifizieren Ressourcen und haben (fast) nichts mit HTTP zu tun. *Browser* arbeiten mit URLs.
- HTML dient zur Beschreibung *formatierter Inhalte* und hat (fast) nichts mit HTTP zu tun.
- Dynamische Ressourcen umfassen ausführbaren Code, der bei einer Anfrage an den Server ausgeführt wird. Über die Standard-Ausgabe wird eine Webseite geliefert.

- *CGI-Programme* können in einer *beliebigen Sprache* formuliert werden und werden in einem schwergewichtigen Prozess ausgeführt. Beliebt sind Skriptsprachen.
- *Servlets* werden in Java programmiert und von einem - JVM-basierten - Server in einem leichtgewichtigen Thread ausgeführt.
- Über den *query string* können textuelle Argumente an CGI-Programme oder Servlets übergeben werden.
- HTML unterstützt die GUI-basierte Eingabe von *query strings* durch *Formulare*.

Quellen

T. Berners-Lee et al. : *Hypertext Transfer Protocol - HTTP/1.1*. RFC 2616.

www.w3.org/Protocols/rfc2616/rfc2616.html

Internet Assigned Numbers Authority: *MIME Media Types*.

www.iana.org/assignments/media-types

T. Berners-Lee et al.: *Uniform Resource Identifiers (URI): Generic Syntax*.

RFC 2396. www.ietf.org/rfc/rfc2396.txt

T. Berners-Lee et al.: *Uniform Resource Locators (URL)*.

RFC 1738. www.faqs.org/rfcs/rfc1738.html

Apache HTTP Server: <http://httpd.apache.org/> <http://hc.apache.org/>

CGI: <http://tools.ietf.org/html/rfc3875>

Anleitung (mit Perl): <http://www.cgi101.com>

Servlets: <http://docs.oracle.com/javaee/6/api/>
<http://tomcat.apache.org>

Formularverarbeitung: <http://www.w3.org/TR/html4/interact/forms.html>

empfehlenswert: *SELFHTML*
<http://de.selfhtml.org/html/>

... sowie die Hinweise auf der Literatur-Seite:

www.inf.fu-berlin.de/lehre/WS12/alp5/literatur.html