Webentwicklung

Backend: HTTP, URLs und Webserver

Inhalt dieser Einheit

1. URLs und Grundlagen
2. HTTP
3. HTTP-Besonderheiten
4. Webserver

Wdh: Was kennen wir schon?

- Bisherige Einheiten:
  - Strukturierung von Inhalten mit **HTML**
  - Trennen von Darstellung und Inhalt mit **CSS**
  - Implementierung von Interaktionsmöglichkeiten mit **JavaScript**
- Ausführung jeweils beim Anwender, im Browser
- Standards wg. geringer Kontrolle über Zielumgebung
  - **W3C** & **WhatWG**: HTML & CSS
  - **ECMA**: JavaScript

Wdh: Nutzerziele erreichen

- Verschiedene Zielarten
  - Ziel: **Zugriff auf Inhalte** (Wikipedia, Youtube, ...)
    - Darstellung von Inhalten: ✓
    - Übertragung von Inhalten: ✗
  - Ziel: **Zugriff auf Anwendungen** (Google Docs, ...)
    - Interaktion im Browser: ✓
    - Übertragung von Anwendungen: ✗
  - Ziel: **Wirkung entfalten** (Social Media, eCommerce, ...)
    - Wirkung außerhalb des eigenen Browsers: ✗
- Offen:
  - Wie gelangen die Inhalte in den Browser?
  - Wie kann eine Wirkung außerhalb des Browsers erzielt werden?

URLs und Grundlagen

Wie kommen Inhalte in den Browser?

- Was machen Sie, um auf eine Website zuzugreifen?
  - Hyperlink folgen, oder
  - Adresse in Browser eintippen

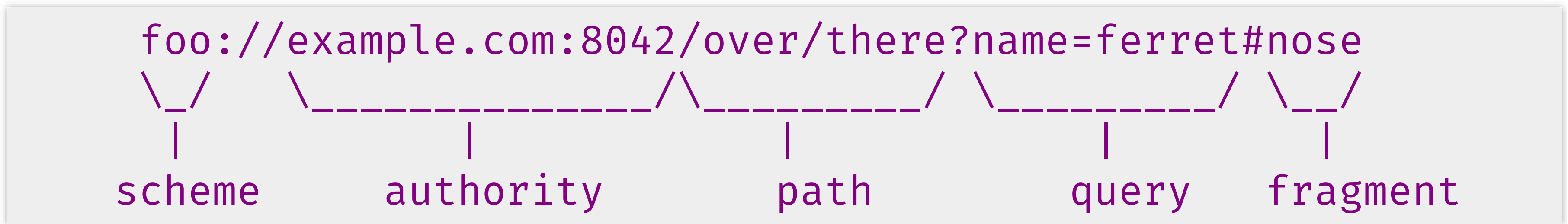
- Wie nennt man solche Adressen?
  - **URL**: *Uniform Resource Locator*
    - Spezialform von **URI**: *Uniform Resource Identifier*

URLs

- Allgemeiner Aufbau:
  - Schema `":"` Schema-spezifischer Teil
- Beispiel-URLs:
  - `http://www.zieris.net/webdev/aufgaben/3#kriterien`
  - `http://www.zieris.net:80/webdev/aufgaben/3`
  - `mailto:zierisf@htw-berlin.de?subject=Frage`
  - `file:///C:/Uni/WebDev/Aufgabe3/index.htm`
- RFC 3986

URL-Standard-Syntax

- Generischer Aufbau
  - Schema `":"` `"//"` Authority Path [`"?"` Query] [`"#"` Fragment]
- Beispiel:



- https://tools.ietf.org/html/rfc3986#page-16
  - Schema: `https`
  - Authority: `tools.ietf.org`
  - Path: `/html/rfc3986`
  - Query: *leer*
  - Fragment: `page-16`

Wie kommen Inhalte in den Browser?

- Was machen Sie, um auf eine Website zuzugreifen?
  - Adresse in Browser eintippen

  - Browser interpretiert Eingabe als URL:
    - Schema: `http`
    - Authority: `www.zieris.net`
    - Path: `/teaching/htw-berlin/webdev/`
    - Query & Fragment: *leer*
- Was passiert dann?
  - → schauen wir mal den Netzwerkverkehr an

Mitschnitt mit Wireshark

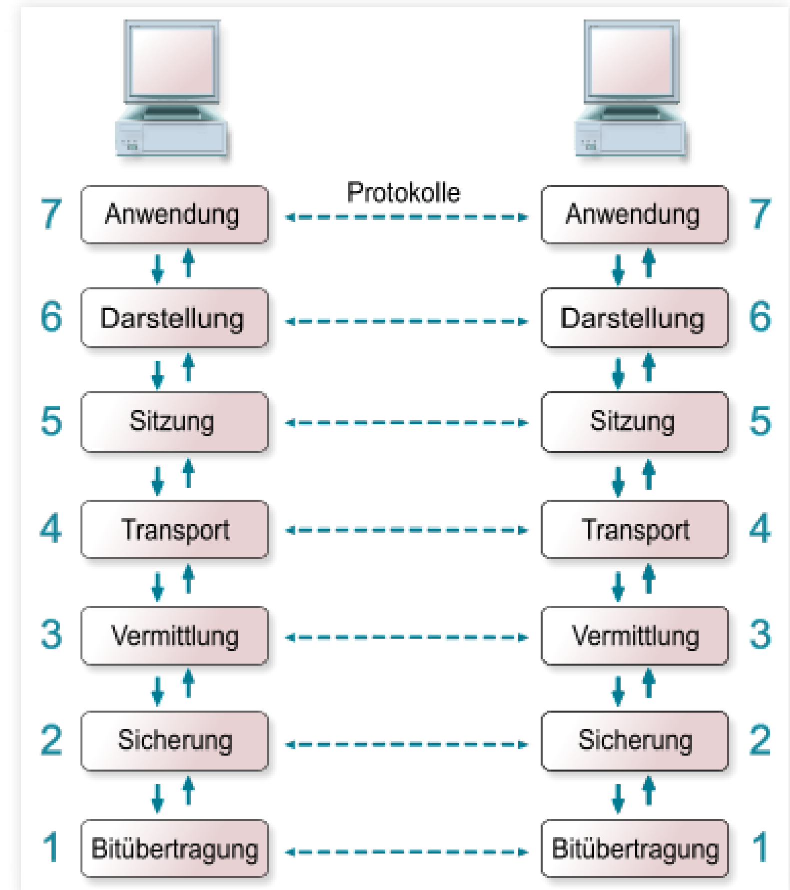
- Mitschnitt von der Netzwerkkarte

Time	Source	Destination	Proto	Len	Info
6.628	192.168.1.93	8.8.8.8	DNS	74	Standard query 0x3eb7 A www.zieris.net
6.695	8.8.8.8	192.168.1.93	DNS	104	Standard query response 0x3eb7 A www.zieris.net
6.696	192.168.1.93	85.214.36.7	TCP	66	49389 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460
6.718	85.214.36.7	192.168.1.93	TCP	66	80 → 49389 [SYN, ACK] Seq=0 Ack=1 Win=14600 Len=
6.718	192.168.1.93	85.214.36.7	TCP	54	49389 → 80 [ACK] Seq=1 Ack=1 Win=66560 Len=0
6.720	192.168.1.93	85.214.36.7	HTTP	541	GET /teaching/htw-berlin/webdev/ HTTP/1.1
6.744	85.214.36.7	192.168.1.93	TCP	60	80 → 49389 [ACK] Seq=1 Ack=488 Win=15744 Len=0
6.768	85.214.36.7	192.168.1.93	TCP	1506	80 → 49389 [ACK] Seq=1 Ack=488 Win=15744 Len=145
6.768	85.214.36.7	192.168.1.93	HTTP	1130	HTTP/1.1 200 OK (text/html)

- 9 Pakete gehen hin und her
  - Warum so viele? Warum nicht nur *Anfrage* und *Antwort*?
  - Grund: Internet und seine Protokolle

OSI-Modell

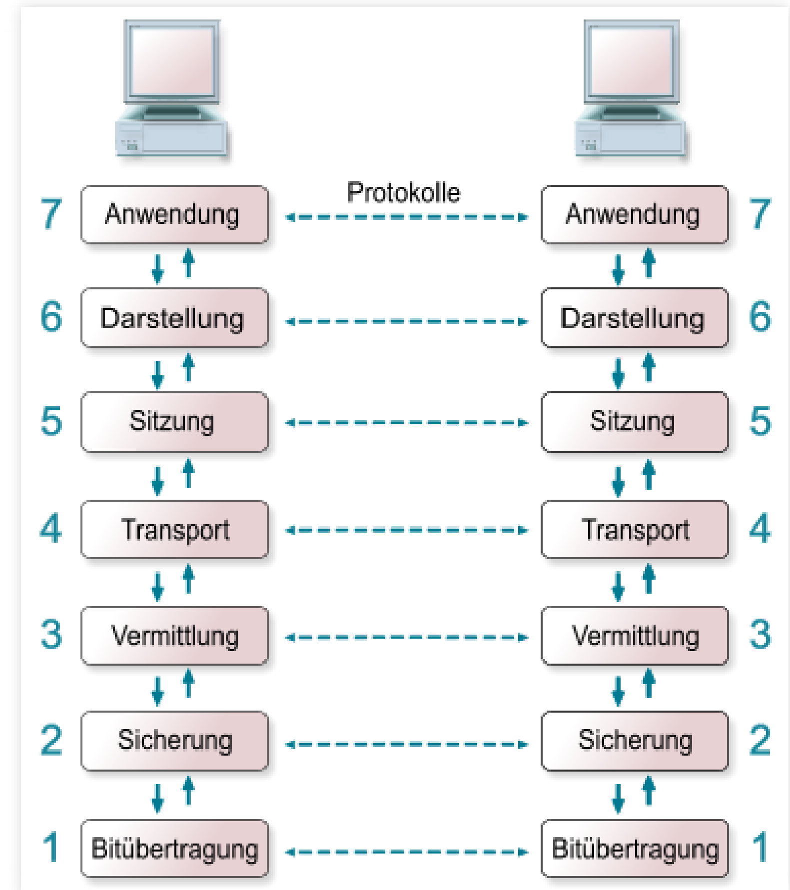
- Blenden Signaltechnik (1) & Ethernet/WLAN (2) aus
- Ebene 3:
  - Kommunikation zweier beliebiger Knoten im Netz
    - Routing, verschiedene Wege
- Protokoll: **IP** (Internet Protocol), IP-Adressen
  - Beispiel: lokal `78.53.3.143`, entfernt `141.45.7.250`



Quelle: http://www.selflinux.org/selflinux/html/osi02.html

OSI-Modell

- Ebene 4:
  - Kommunikation zweier Prozesse
    - zuverlässige Verbindung, Stauvermeidung
- Protokoll: **TCP** (*Transmission Control Protocol*), Ports
  - Beispiel: lokal `64925`, entfernt `80`



Quelle: http://www.selflinux.org/selflinux/html/osi02.html

OSI-Modell

- Ebenen 5-7: Anwendungen
  - Problem: IP-Adressen lassen sich schwer merken
- Protokoll: **DNS** (*Domain Name System*), Records
  - Beispiel:

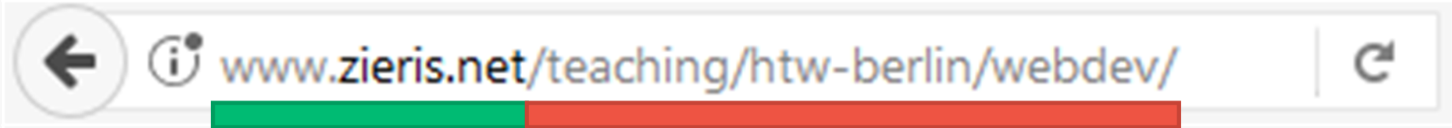
```
www.htw-berlin.de.      IN   CNAME berndcms.htw-berlin.de
berndcms.htw-berlin.de. IN   A     141.45.7.250
```

Mitschnitt mit Wireshark

Time	Source	Destination	Proto	Len	Info
6.628	192.168.1.93	8.8.8.8	DNS	74	Standard query 0x3eb7 A www.zieris.net
6.695	8.8.8.8	192.168.1.93	DNS	104	Standard query response 0x3eb7 A www.zieris.net
6.696	192.168.1.93	85.214.36.7	TCP	66	49389 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460
6.718	85.214.36.7	192.168.1.93	TCP	66	80 → 49389 [SYN, ACK] Seq=0 Ack=1 Win=14600 Len=
6.718	192.168.1.93	85.214.36.7	TCP	54	49389 → 80 [ACK] Seq=1 Ack=1 Win=66560 Len=0
6.720	192.168.1.93	85.214.36.7	HTTP	541	GET /teaching/htw-berlin/webdev/ HTTP/1.1
6.744	85.214.36.7	192.168.1.93	TCP	60	80 → 49389 [ACK] Seq=1 Ack=488 Win=15744 Len=0
6.768	85.214.36.7	192.168.1.93	TCP	1506	80 → 49389 [ACK] Seq=1 Ack=488 Win=15744 Len=145
6.768	85.214.36.7	192.168.1.93	HTTP	1130	HTTP/1.1 200 OK (text/html)

- 9 IP-Pakete
  1. DNS: Domain in IP-Adresse auflösen
  2. TCP: Verbindung herstellen (Drei-Wege-Handshake)
  3. HTTP: Anfrage der Resource, [TCP-Bestätigung] und Antwort [in zwei TCP-Segmenten]

Anfrage schematisch



0ms **DNS**: Query
www.zieris.net: type A, class IN

67ms **DNS**: Answer
www.zieris.net: type CNAME, cname zieris.net
zieris.net: type A, addr 85.214.36.7

68-90ms **TCP:** Handshake mit 85.214.36.7 auf Port 80

92ms

```
GET /teaching/htw-berlin/webdev/ HTTP/1.1
Host: www.zieris.net
User-Agent: Mozilla/5.0 ...
```

116ms **TCP:** ACK

140ms **TCP:** Seg. 1

141ms **TCP:** Seg. 2

```
HTTP/1.1 200 OK
Date: Fri, 10 Nov 2017 16:25:17 GMT
Server: Apache
Content-Length: 2267
Content-Type: text/html; charset=UTF-8

<!DOCTYPE html>
<html lang="de"> ...
```

Aufgaben der Protokolle

- Aufgabenverteilung
  1. Domänenname muss aufgelöst werden (DNS)
  2. Verbindung muss aufgebaut werden (TCP)
  3. Übertragung von Dokumenten/Ressourcen muss ermöglicht werden (HTTP)
     - (TCP sorgt für reibungslosen Ablauf)
- Heute:
  - Ignorieren 1 und 2 (werden später noch mal interessant)
    - Merke: stellen Kommunikationskanal, um mit Webserver zu reden
  - Schauen uns HTTP genauer an

HTTP

HTTP

- *Hypertext Transmission Protocol*:
  - Protokoll zur Übertragung von Hypertext, d.h. Ressourcen wie Texte, Bilder, später dann auch CSS- und JavaScript-Dateien
- Kommunikation über Nachrichten, zustandslos:
  - Client sendet Anfrage (Request) an Server
  - Server antwortet (Response)
- Format: ASCII, also leicht zu schreiben und zu lesen
  - (TCP und IP sind platzsparende Binärformate)

HTTP-Nachrichten-Aufbau

- Aufbau von Request und Response:
  - **Start Line**
  - **Header** (evtl. leer)
  - Leerzeile
  - **Body** (evtl. leer)
- (nicht verwechseln mit HTML-`head` und -`body`)

Einfache HTTP-Anfrage

- Nachricht:

```
GET /index.htm HTTP/1.1
```

- Bestandteile (Start Line):
  - Method: `GET`
  - Request Target: `/index.htm`
  - Version: `HTTP/1.1`
- sonstiger Header und Body: *leer*

Einfache HTTP-Antwort

- Nachricht:

```
HTTP/1.1 200 OK
Content-type: text/html; charset=utf-8
Content-length: 2014

<!DOCTYPE html>
<html lang="de">
  <head>
  ...
```

- Bestandteile (Start Line):
  - Version: `HTTP/1.1`
  - Status Code und Message: `200` `OK`
- Header: Inhaltsart (MIME-Type) und -länge
- Body: Das HTML-Dokument

Antwort: Status Codes

Code	Gruppe	Bedeutung
`1xx`	Informational	Anfrage-Bearbeitung dauert noch
`2xx`	Success	Anfrage war erfolgreich
`3xx`	Redirection	Bearbeitung erfordert weitere Schritte des Clients
`4xx`	Client Error	Problem liegt wohl bei Client
`5xx`	Server Error	Problem liegt wohl bei Server

- Alle: List of HTTP status codes
- Beispiele folgen …

Antwort: häufige Status Codes

Code	Nachricht	Bedeutung
`200`	`OK`	Anfrage erfolgreich beendet
`201`	`Created`	Alles gut, Ressource angelegt.
`301`	`Moved Permanently`	Adresse nicht mehr gültig, Umleitung zu neuer Location
`307`	`Temporary Redirect`	Vorübergehend woanders vorhanden

Antwort: häufige Status Codes

Code	Nachricht	Bedeutung
`401`	`Unauthorized`	Authentifizierung erfordert
`403`	`Forbidden`	Client ist nicht berechtigt
`404`	`Not Found`	Ressource wurde nicht gefunden
`500`	`Internal Server Error`	Irgendwas ging auf dem Server schief
`503`	`Service Unavailable`	Server eventuell überlastet

Antwort: Header

- wenn der Server noch mehr mitteilen möchte:

```
HTTP/1.1 200 OK
Date: Mon, 06 Oct 2014 16:07:02 GMT
Server: Apache
X-Powered-By: PHP/5.3.15
Cache-Control: private
Content-Length: 3253
Connection: Keep-Alive
Content-Type: text/html; charset=utf-8
```

- Viel mehr:
  - https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers

Antwort: MIME-Type

- `Content-Type` gibt Art von Inhalt an
  - Beispiel: gleicher Inhalt, `text/plain` vs. `text/html`

- Mehr dazu:
  - https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types

Anfrage: Methode

- Am häufigsten benutzt:
  - `GET`: Ressourcen vom Webserver anfordern
  - `POST`: Inhalte an den Webserver senden
    - z.B. Inhalt eines Textfeldes bei Webmail
-

```
POST /kontakt HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 61

to=zierisf@htw-berlin.de&subject=Hallo&content=Test-Nachricht
```

```
<form action="/kontakt" method="post">
  <input name="to"><input name="subject">
  <textarea name="content"></textarea>
  <input type="submit">
</form>
```

Anfrage: HTTP-Methoden

Methode	Zweck
`GET`	Anfragen einer Ressource
`POST`	Anlegen einer Ressource (oder ändern)
`HEAD`	Nur Header, nicht Body anfragen.
`PUT`	Ersetzt eine Ressource (oder legt sie an)
`DELETE`	Löschen einer Ressource
`OPTIONS`	Liste der unterstützten Methoden

- Details und weitere Methoden:
  - https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

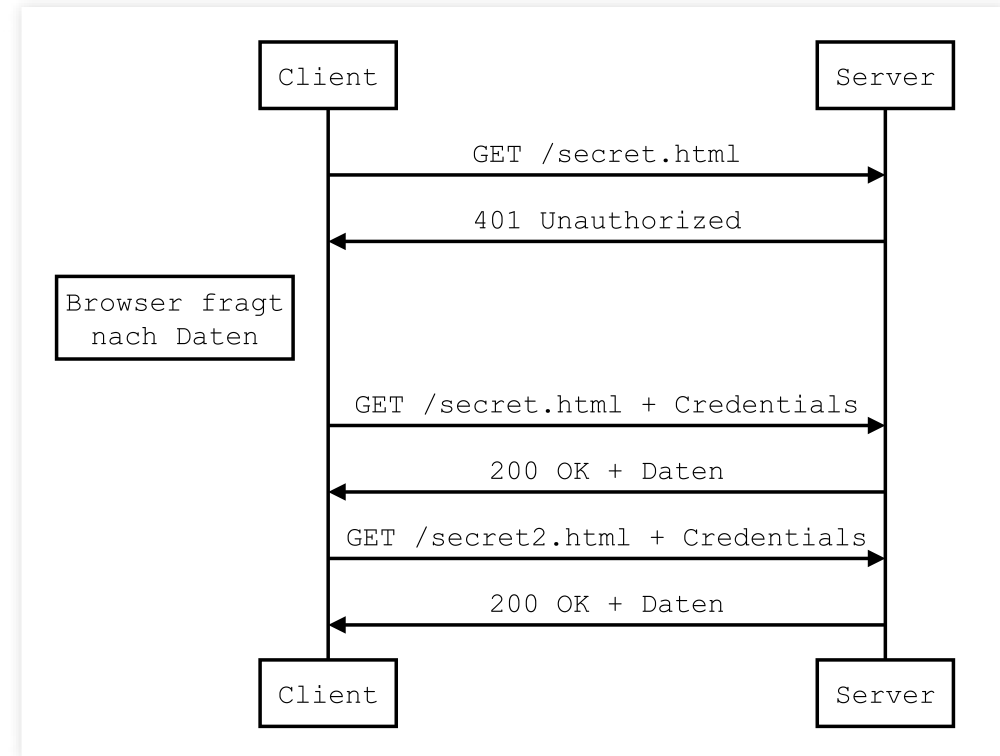
Aufgaben eines Webservers

- (technisch betrachtet)
- TCP-Verbindungen (default: Port 80) entgegennehmen
- Auf HTTP-Anfragen mit HTTP-Antworten reagieren
  - Berücksichtigen von: Anfrage-Methode, Anfrage-Ziel, zusätzliche Header-Zeilen und Anfrage-Body
- um so Ressourcen bereitzustellen

HTTP-Besonderheiten

Anwendungsfall: Inhalte schützen

- Authentifizierung muss serverseitig konfiguriert werden
- Verfahren
  - *Basic Authentication*
  - *Digest Access Authentication*



Beispiel: Basic Authentication

```
GET /secret.html HTTP/1.1
```

```
HTTP/1.1 401 Authorization Required
Server: Apache/2.2.16 (Debian)
WWW-Authenticate: Basic realm="Authorized personnel only."
```

```
GET /secret.html HTTP/1.1
Authorization: Basic YWRtaW46eW9sbw==
```

- Details:
  -

    ```
    base64_decode('YWRtaW46eW9sbw==') == 'admin:yolo'
    ```
  - https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication#Basic_authentication_schem

Anwendungsfall: Nutzer erkennen

- HTTP zustandslos → erschwert Nutzer-Indentifizierung
- Entwicklung der "Magic Cookies" durch Netscape
  - Älteste und weitest verbreitete Speichermöglichkeit im Client
  - Setzen durch Response-Header

```
Set-Cookie: value[; expires=date][; domain=name][; path=path]
  [; secure][; HttpOnly]
```

  - Werden dann bei jeder HTTP-Anfrage mitgeschickt

Beispiel: Cookies

```
GET / HTTP/1.1

```

```
HTTP/1.1 200 OK
Set-Cookie: datr=cQw8VCSHQe2BlCBjtE4JvYOa;
   expires=Wed, 12-Oct-2016 17:31:55 GMT; Max-Age=63072000;
   path=/; domain=.facebook.com; httponly
Set-Cookie: reg_ext_ref=deleted;
   expires=Thu, 01-Jan-1970 00:00:01 GMT;
   Max-Age=0; path=/; domain=.facebook.com
Set-Cookie: reg_fb_gate=https%3A%2F%2Fwww.facebook.com%2F;
   path=/; domain=.facebook.com
```

```
GET / HTTP/1.1
Cookie: datr=cQw8VCSHQe2BlCBjtE4JvYOa;
   reg_fb_gate=https%3A%2F%2Fwww.facebook.com%2F
```

- Details:
  - https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies

HTTP-Debugging: cURL

```
$ curl -v htw-berlin.de/
```

```
GET / HTTP/1.1
Host: htw-berlin.de
User-Agent: curl/7.47.1
Accept: */*
```

```
HTTP/1.1 302 FOUND
Date: Tue, 26 Sep 2017 15:17:56 GMT
Server: Apache
Location: http://www.htw-berlin.de/
Content-Length: 0
Content-Type: text/plain; charset=ISO-8859-1
```

- Eine Weiterleitung an eine andere `Location`
- `Host`: mehrere Domains für eine IP-Adresse möglich

HTTP-Debugging: cURL

```
$ curl -v www.htw-berlin.de/
```

```
GET / HTTP/1.1
Host: www.htw-berlin.de
User-Agent: curl/7.47.1
Accept: */*
```

```
HTTP/1.1 200 OK
Server: Apache
Cache-Control: private
Content-Length: 35835
Content-Type: text/html; charset=utf-8

<!DOCTYPE html>
<html lang="de">
<head>
...
```

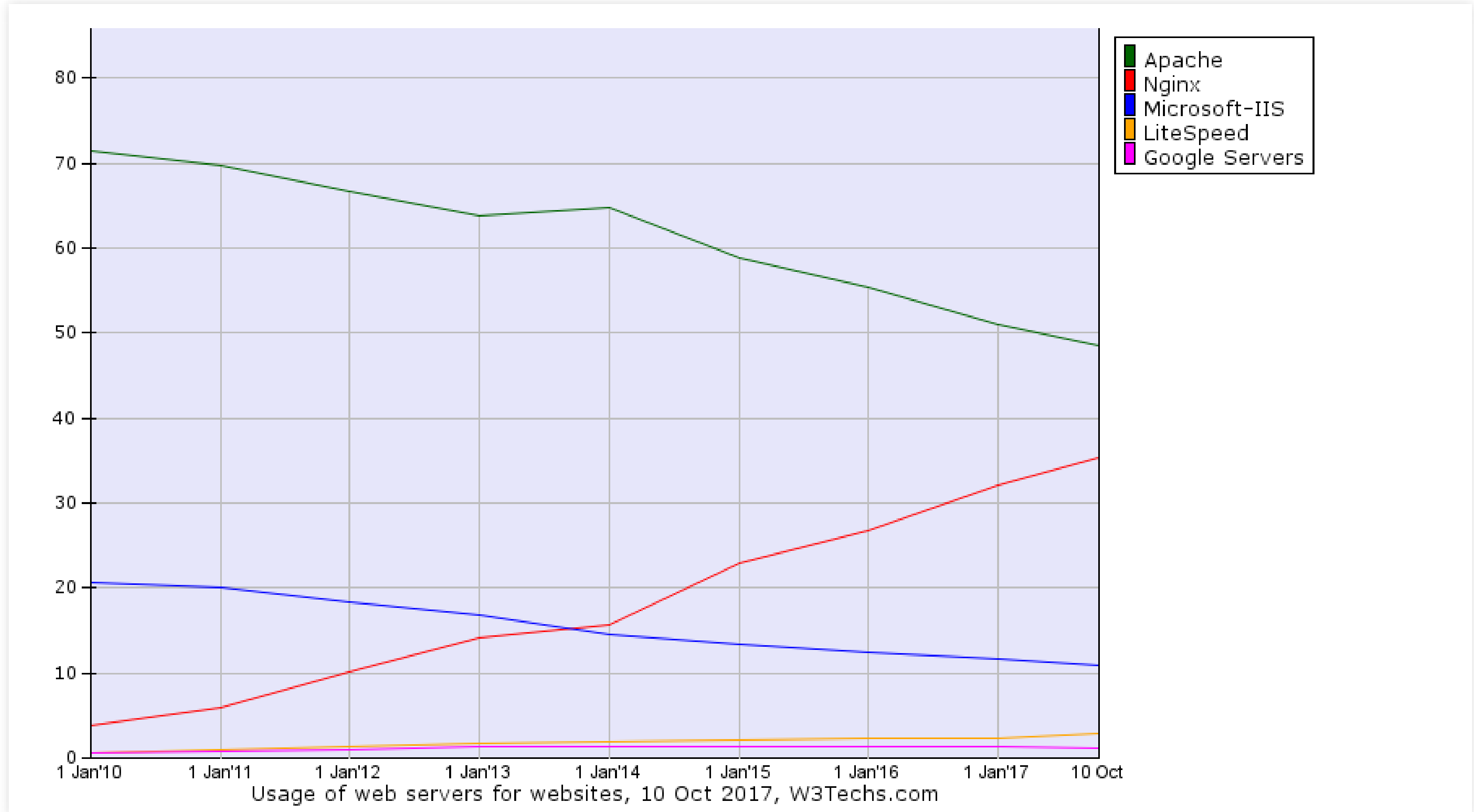
- Dokument wird geliefert, obwohl nicht explizit genannt

Webserver

Unterschiedliche Request Targets

- **statische Ressourcen**:
  - HTML-Dokumente, CSS-Stylesheets, JavaScript-Dateien
  - Request-Ziel ist eine Datei im Dateisystem des Servers
  - Dateiinhalt wird bei `GET`-Requests ausgeliefert
- **dynamische Ressourcen**:
  - Client-Anfragen (`POST`, `PUT`, `DELETE`, …) sollen Wirkung entfalten
  - Request-Ziel ist ein Programm
  - wird vom Webserver beim Request angesprochen/gestartet
  - Programmausgabe wird als Response gesendet
- Aus Sicht von HTTP kein Unterschied!
  - (`GET`-Anfragen können von Programmen beantwortet werden; `POST`s können an statische Dateien gehen – ohne Effekt)

Konkrete Webserver



Quelle: https://w3techs.com/technologies/history_overview/web_server/ms/y

Konkreter Webserver: Apache

- Korrekt *Apache HTTP Server*
  - https://httpd.apache.org/docs/2.4/
- Weit verbreitet
  - Auf Laborrechnern installiert
  - Eigener Rechner: XAMPP
- Umfangreiche Konfigurations- und Erweiterungsmöglichkeiten
  - z.B. `VirtualHost`s: mehrere Domains, mehrere IP-Adressen in einem Dienst

Apache-Konfiguration

- Dienst-Konfiguration über *Direktiven* in Text-Dateien
  - Basis-Datei (z.B. `httpd.conf`)
  - Einbindung weiterer Dateien über `Include`-Direktive
  - Direktiven werden beim Starten des Dienstes angewendet
    - Änderungen: Neustart
- zusätzlich: `.htaccess`-Dateien
  - liegen im gleichen Ordner wie die Website-Dateien
  - können nur bestimmte Direktiven beinhalten
  - Direktiven werden bei jedem Request angewendet
    - Änderungen: Seite neu laden, kein Neustart nötig

Apache: Beispiel-Konfiguration

```
# Auf Verbindungen an Port 80 reagieren
Listen 80

# Alle Dateien liegen unterhalb dieses Verzeichnisses
DocumentRoot /usr/web

DirectoryIndex index.html index.htm
AddType text/html html htm
```

- Anfrage: `GET / HTTP/1.1`
  - falls Datei existiert, liefert den Inhalt von `/usr/web/index.html` aus
    - (sonst Fallback auf `/usr/web/index.htm`)
  - mit Response-Header `Content-Type: text/html`
- Details: https://httpd.apache.org/docs/2.4/

Statische Ressourcen: Direktiven

- Mapping von Request-Targets auf Dateien-System
  - `DocumentRoot`: Basis für alle Request-Targets, relative Pfade
  - `Alias`: Ressource unter anderem Namen verfügbar machen

    ```
    Alias "/docs" "/var/web"
    ```

  - `Redirect`: Client an andere Stelle lenken

    ```
    Redirect "/webdev/" "http://zieris.net/teaching/htw-berlin/webdev/"
    ```

  - und viele mehr (reguläre Ausdrücke, …)
- Details:
  - https://httpd.apache.org/docs/2.4/urlmapping.html

Dynamische Ressourcen

- Apache (und andere Webserver) bietet eine Umgebung für Programme, kümmert sich um:
  - Netzwerkschicht (TCP/IP)
  - URL-Mapping (siehe oben)
  - Auswertung der Request-Start Line und einiger Header
  - Setzen der Response-Start Line und einiger Header
- Verschiedene Arten um Programme anzusprechen
  - CGI (Common Gateway Interface)
  - Sprach-spezifische Module

Dynamische Ressourcen über CGI

- CGI: Common Gateway Interface
  - Bei jeder Anfrage wird ein neuer Prozess gestartet
  - Informationen aus der Anfrage (Header und Body) werden als Umgebungsvariable bereitgestellt
  - Programm muss Ausgabe mit `Content-Type: ...` erzeugen (Leerzeile nicht vergessen)

Dynamische Ressourcen über CGI

```
# CGI-Ausführung im Apache aktivieren
LoadModule cgi_module modules/mod_cgi.so

# Anfragen an '/cgi-bin/' an Ordner mit CGI-Skripten umleiten
ScriptAlias /cgi-bin/ /path/to/scripts/
```

```
#!/usr/bin/bash
# Ein beliebiges Shell-Skript im CGI-Ordner
echo "Content-Type: text/plain\n\nHallo, $HTTP_USER_AGENT"
```

```
#!/usr/bin/python
# oder Python
print 'Content-Type: text/plain\n\nHallo, Welt!'
```

```
#!/usr/local/bin/runhugs
-- oder wie wäre es mit Haskell?
```

PHP und Apache: händisch via CGI

```
#!/usr/bin/php
<?php
// Datei im CGI-Ordner
echo "Content-Type: text/html\n\n";
echo "Hallo, " . $_ENV["HTTP_USER_AGENT"];
?>
```

- Request-Target: Skript, das PHP-Interpreter aufruft
  - Skript verweist auf PHP-CLI-Interpreter (Zeile 1)
    - und muss HTTP-Content-Typ selbst angeben
- Request-Eigenschaften sind als Umgebungsvariablen verfügbar
- Interpreter-Prozess wird für jede Anfrage gestartet
- **macht kein Mensch**

PHP und Apache: PHP-CGI-Modul

```
# Dateien mit .php-Endung an PHP-CGI-Modul weiterreichen
<FilesMatch \.php$>
    SetHandler application/x-httpd-php-cgi
</FilesMatch>

# Angeben eines Executables
Action application/x-httpd-php-cgi /path/to/php-cgi-binary
```

- Request-Target: PHP-Datei, die an Interpreter übergeben wird
- PHP-CGI ruft den Interpreter auf
  - und fügt HTTP-Content-Typ hinzu
- Interpreter-Prozess wird für jede Anfrage gestartet
- **macht man manchmal**

PHP und Apache: Als Modul

```
# SAPI-Modul in den Webserver integrieren
LoadModule php7_module modules/libphp7.so

# Dateien mit .php-Endung an PHP zur Interpretation weiterreichen
<FilesMatch \.php$>
    SetHandler application/x-httpd-php
</FilesMatch>
```

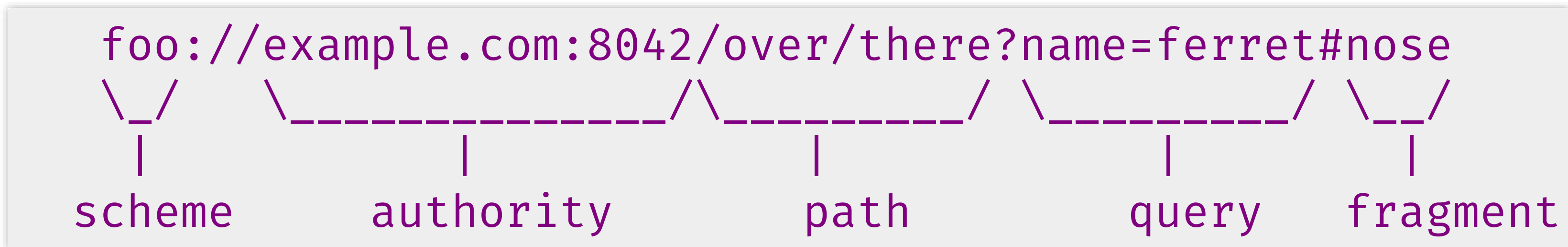
- Request-Target: PHP-Datei, die an den Interpreter übergeben wird
- Starten des PHP-Prozesses mit Webserver-Start
  - deutlich schneller als CGI
- **macht man normalerweise**

Dynamische Ressourcen: Eingaben

- Programme: Eingabe -> Verarbeitung -> Ausgabe
  - Format Eingabe: **HTTP**
  - Format Ausgabe: **HTTP** (darin HTML, CSS, JavaScript)
  - Verarbeitung: *ab nächster Woche*
  - Eingabe-Parameter: **??**
- Parameter:
  - zwei Möglichkeiten: `GET` und `POST`-Parameter

GET-Parameter

- URI-Syntax:



- Bedeutet: Variable `name` hat den Wert `ferret`
- Parameter sind Teil der URL
  - Damit auch Teil der Browser-Historie
  - (Fragment wird *nicht* an den Server übertragen)

POST-Parameter

- abgeschicktes Formular:

```
POST /kontakt HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 61

to=zierisf@htw-berlin.de&subject=Hallo&content=Test-Nachricht
```

- Parameter sind *nicht* Teil der URL
  - Landen damit auch nicht in der Browser-Historie

Mischform & Wann was?

- möglich:

```
POST /kontakt?action=send HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 61

to=zierisf@htw-berlin.de&subject=Hallo&content=Test-Nachricht
```

- nicht gut:

```
POST /login?username=peter&password=peteristderbeste HTTP/1.1
```

- besser:

```
POST /login HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 40

username=peter&password=peteristderbeste
```

Apache: Wartung und Debugging

- Apache schreibt Logfiles
  - `access.log`

    ```
    78.154.4.85 - frank [10/Oct/2017:13:55:36 +0200]
      "GET /index.htm HTTP/1.1" 200 2326
    ```

  - `error.log`

    ```
    [Fri Sep 09 10:42:29.902022 2011] [core:error]
      [pid 35708:tid 4328636416] [client 78.154.4.85]
      File does not exist: /usr/local/apache2/htdocs/favicon.ico
    ```

- GET-Parameter landen im Access-Log, POST-Parameter nicht
- Mehr dazu:
  - https://httpd.apache.org/docs/2.4/logs.html

Eigener Webserver: Heroku

- Heroku: Hosting von Webanwendungen
  - Verschiedene Webserver, z.B. Apache – Basiskonfiguration, anpassbar
  - Verschiedene Programmiersprachen, z.B. PHP, Java, Ruby, Python, Node.js
  - Freemium-Model: kostenlos für einfache Zwecke
  - Deployment über `git push`
- Werden wir ab Übung 4 benutzen

Zusammenfassung: Heutige Einheit

- Eigenschaften des HTTP-Protokolls
  - ASCII-basiert
  - Header und Body
  - Request und Response
  - Zustandslos
- URLs
- Webserver
  - bearbeitet Requests
  - statische und dynamische Ressourcen
  - erzeugt Responses
- Grundlegendes zur Webserver-Konfiguration
- Anwendungen: CGI vs. Module

Danke!