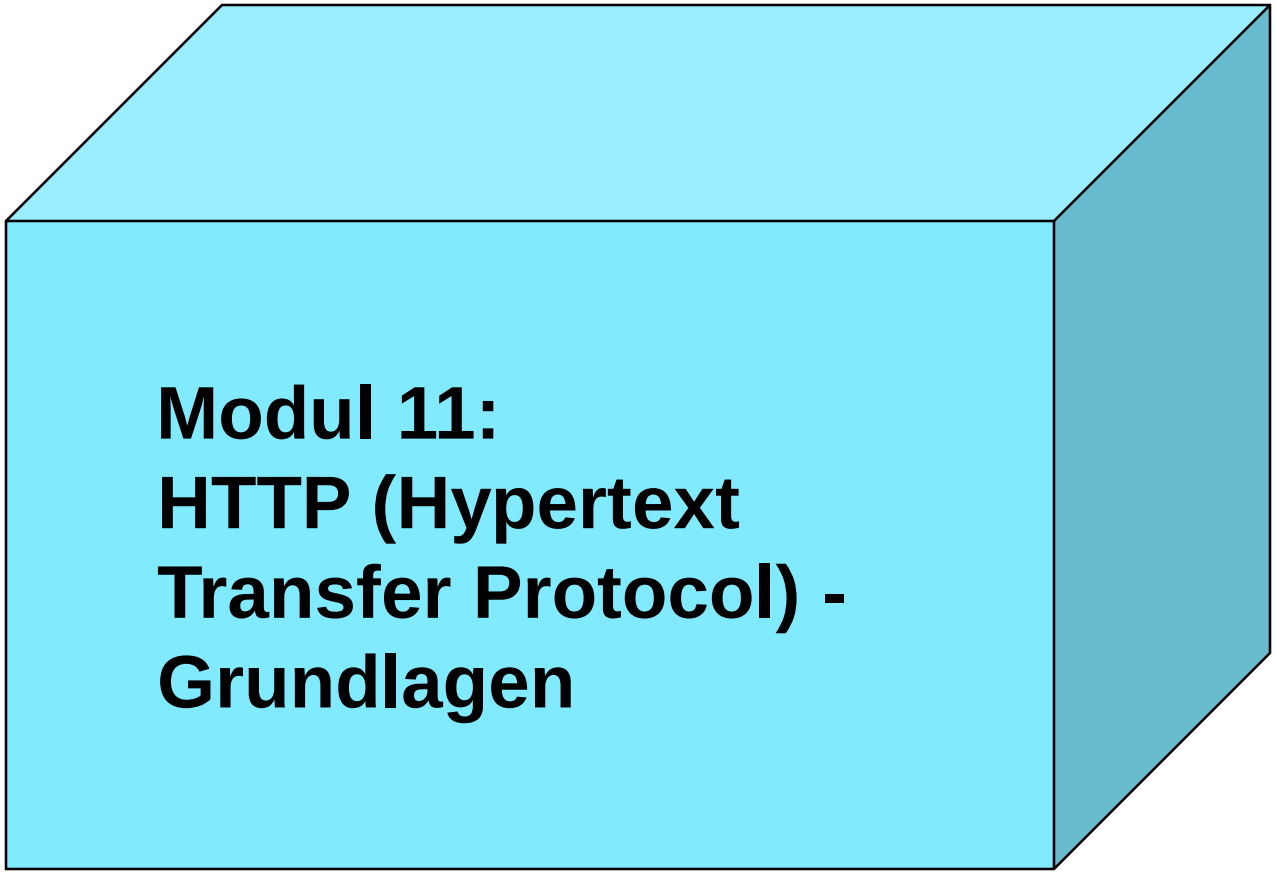


HTTP-Grundkonzept

- HTTP setzt auf das Protokoll TCP auf.
- Der Web-Server benutzt standardmäßig den TCP-Port 80.
- HTTP ist ein zustandsloses Request/Response-Protokoll.
- HTTP ist textorientiert.
- Typische Methoden sind
  - GET: Abruf des in der URL bezeichneten Dokuments
  - HEAD: Abruf von Metainformationen über das in der URL bezeichnete Dokument
  - …
- Ein HTTP-Server muss nur die Methoden GET und HEAD implementieren.
- Designziele von HTTP sind:
  - Einfachheit: Das Protokoll soll nur wenige Ressourcen beanspruchen und einfach zu implementieren sein. (Davon entfernen wir uns gegenwärtig wieder.)
  - Geschwindigkeit: Das Protokoll soll so schnell wie möglich sein.

Syntax HTTP Request Messages

Ein HTTP Request Messages ist eine strukturierte Folge von lesbaren ASCII Zeichen.

Strukturierung gemäß RFC 2616 (5.1) wie folgt:

```
Request = Request-Line

          *(( general-header
              | request-header
              | entity-header ) CRLF)

          CRLF

          [ message-body ]


Request-Line   = Method SP Request-URI SP HTTP-Version CRLF
```

"The Request-URI is a Uniform Resource Identifier and identifies the resource upon which to apply the request. " Hier sind verschiedene Formen möglich.

Beispiel HTTP Request Messages

```
GET / HTTP/1.1

Host: docklab.de:9991
Accept-Encoding: gzip, deflate
Accept-Language: de,en-US;q=0.7,en;q=0.3
Accept: text/html,application/xml;q=0.9,image/webp,*/*;q=0.8
Connection: keep-alive
Dnt: 1
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:88.0)
Gecko/20100101 Firefox/88.0
```

Syntax HTTP Response Messages

Ein HTTP Request Messages ist eine strukturierte Folge von lesbaren ASCII Zeichen.

Strukturierung gemäß RFC 2616 wie folgt:

```
Response  = Status-Line

            *(( general-header
                | response-header
                | entity-header ) CRLF)

            CRLF

            [ message-body ]
```

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

"The first digit of the Status-Code defines the class of response. " Hier sind verschiedene Fälle möglich.

Beispiel HTTP Response Messages

```
HTTP/1.1 200 ok

Content-Type: text/plain
X-Real-Server: echo-server
Date: Wed, 12 May 2021 17:37:35 GMT
Content-Length: 2011

Welcome to echo-server!  Here's what I know.
  > Head to /ws for interactive websocket echo!

-> My hostname is: 91effe566693

-> Requesting IP: 94.134.44.189:49532

-> Request Headers |

  HTTP/1.1 GET /

  Host: docklab.de:9991
  Accept-Encoding: gzip, deflate, br
```

Mögliche Eigenschaften einer (konkreten) HTTP-Methode

- **sicher:**
  Die Semantik der Methode ist read-only. Das Ausführen einer sicheren Methode hat keine Seiteneffekte oder Zustandsänderungen.
  Sollte die URL beim Ausführen einer sicheren Methode eine Aktion bewirken, so liegt dies in der Verantwortung des Ressorcenbesitzers.

- **idempotent:**
  Wiederholte Ausführung der Methode hat die gleiche Wirkung wie die einmalige Ausführung.
  "Mehrmaliges Anklicken eines Links ist explizit erlaubt", da GET idempotent.

- **cacheable:**
  Um die Performanz von HTTP zu steigern, wird im Internet gerne Caching eingesetzt. Diese Funktionalität wird insbesondere von Content Delivery Network Providern (CDN-Provider, z.B. Cloudflare) bereitgestellt.

HTTP-Methoden

Methode	Beschreibung
GET	Anforderung einer Ressource. Die Ressource wird in der URL spezifiziert. Die URL kann komplex sein (→ REST). GET wird von allen Browsern unterstützt. Mit GET können über die URL auch Daten zum Server übertragen werden. GET ist sicher, idempotent und cacheable. Ohne Body. Beispiel: GET /search?platform=Linux&category=media

POST	Wird verwendet, um Daten an den Server zu senden. Hauptanwendung Onlineformulare. Format der Daten wird im Attribut Content-Type: beschrieben. Die Daten werden im Body übergeben. Beispiel:
	`Content-Type: application/x-www-form-urlencoded`
	
	`vorname=martin&nachname=leischner`
	POST ist nicht sicher, nicht idempotent, bedingt cacheable (d.h. nur mit freshness Info). Mit Body.

HEAD	Gibt nur die Header zurück, die im Fall eines GET-Requests verwendet worden wären. Client kann dann entscheiden, wie er weiter vorgehen will. HEAD ist sicher, idempotent und cacheable. Ohne Body. Beispiel: HEAD /downloads/video.mpeg HTTP/1.1

HTTP-Methoden

Methode	Beschreibung
PUT	Kreiert eine neue Ressource oder ersetzt eine vorhandene. Die Daten werden im Body übergeben. Erfolgreiche Antwort (Code 200 oder 204) enthält keinen Body. Das neue Kreieren einer Ressource wird durch den Code 201 (Created) angezeigt. PUT ist nicht sicher, idempotent, nicht cacheable. Mit Body. Beispiel: PUT /new.html HTTP/1.1
OPTIONS	Fragt den Server nach den zugelassenen HTTP-Methoden. Hierbei kann eine spezielle URL oder für den ganzen Server das Sternsymbol * angegeben werden. Die Optionen finden sich im Response Header *Allow*. Die Methode OPTIONS wird oft deaktiviert. OPTIONS ist sicher, idempotent, aber nicht cacheable. Ohne Body. Beispiel: OPTIONS http://dh.docklab.de:60000 HTTP/1.1
DELETE	Löscht die angegebene Ressource. Wird für API-Server verwendet. Ist bei contentorientierten Servern abgeschaltet. DELETE ist nicht sicher, idempotent, aber nicht cacheable. Body möglich. Beispiel:
TRACE	Loop-Back-Test. Request wird so zurückgeschickt, wie er empfangen wurde. TRACE ist sicher, idempotent, aber nicht cacheable. Ohne Body. Beispiel:

HTTP-Spezialmethoden

Methode	Beschreibung
PATCH	Während die Methode PUT eine Ressource komplett ersetzt, kann mit PATCH eine Ressource gezielt modifiziert werden. Die im Body spezifizierten Modifikationen müssen auf die Ressource ganz oder gar nicht angewendet werden ("atomar"). Die Modifikationen können z.B. über einen JSON PATCH (RFC6902) syntaktisch korrekt im Body angegeben werden. PATCH ist nicht sicher, nicht idempotent, nicht cacheable. Mit Body. Der Body der Response enthält eine Repräsentation der geänderten Ressource.
CONNECT	Durch CONNECT wird ein Tunnel zur angefragten Ressource aufgebaut. Danach können über den Tunnel "blind" Daten ausgetauscht werden, bis der Tunnel geschlossen wird. CONNECT soll nur für Anfragen an einen Proxy verwendet werden. CONNECT ist nicht sicher, nicht idempotent, nicht cacheable. Ohne Body. Beispiel:

HTTP Response Status Codes

Es gibt eine Vielzahl von Status Codes. Die wichtigsten:

- **HTTP Status Code 200 – OK**
- **HTTP Status Code 301 - Moved Permanently:**
  Ressource hat andere URL. Diese findet sich im Location Header.
- **Status-Code 302 – Moved Temporarily**
- **Status-Code 403 – Forbidden**
- **HTTP Status Code 404 - Not Found:**
  Weiterleitung auf Fehlerseite oder auf eine Most-Relevant-Seite. Schlecht ist die Weiterleitung auf die Homepage, da dies verwirrend wirkt.
- **Status-Code 500 – Internal Server Error:**
  Server ist abgestürzt.
- **HTTP Status Code 503 - Service Unavailable:**
  Server ist überlastet. Hilfreich ist hier eine Angabe darüber, wann der Server voraussichtlich wieder erreichbar ist.

Identifikation von Ressourcen (heute übliches Schema)

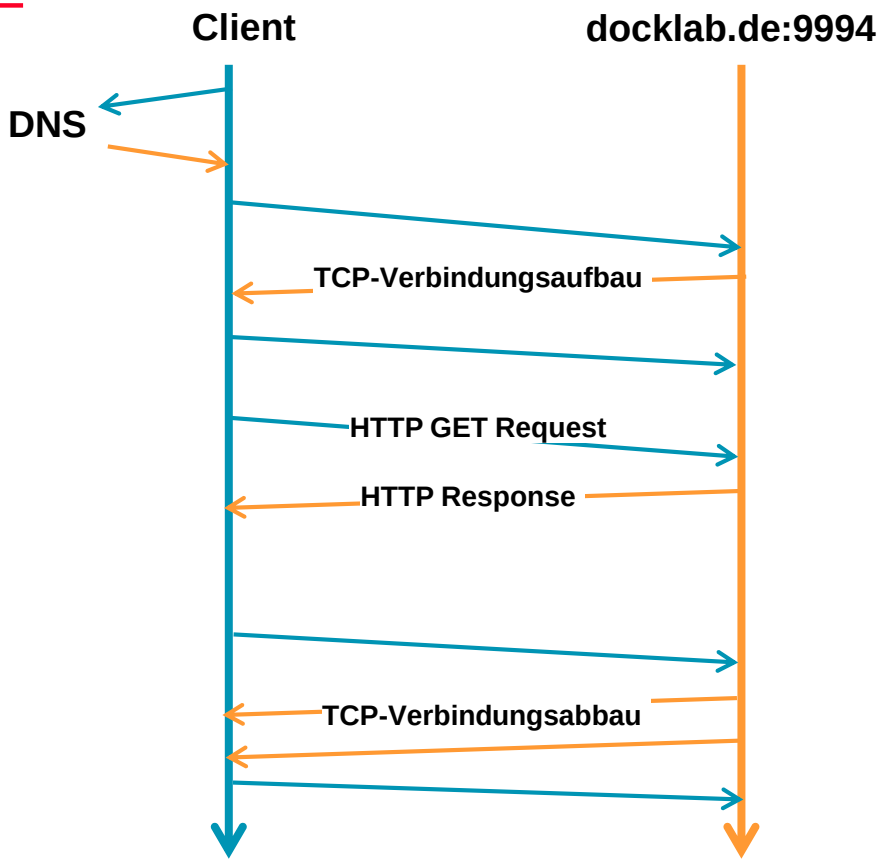
http://www.example.com:80/path/to/myfile.html?key1=value1&key2=value2#SomewhereInTheDocument

Bestandteil	Name	Andere Beispiele, Bemerkung
http	Protokoll	https, mailto, ftp, file
www.example.com	Authority	192.34.23.8
80	Port	Kann entfallen oder andere Portnummern
path/to/myfile.html	Pfad	Ist heute meist ein 'abstrakter' Pfad (routing)
key1=value1&key2=value2	Query	Syntax ist nicht festgelegt, serverspezifisch
SomewhereInTheDocument	Fragment	Zeiger in die Ressource, wird vom Browser gewöhnlich nicht an Server gesendet.

Grundsätzlicher Ablauf eines HTTP-Request-Response

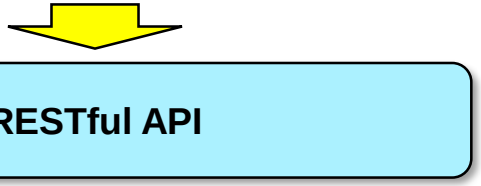
Beispiel: Aufruf von http://docklab.de:9994/

1. **DNS-Auflösung**
2. **Aufbau einer TCP-Verbindung zu 194.95.66.107:9994**
3. **HTTP GET Request Messages als Bytestrome an 194.95.66.107:9994 senden. Bei Body recht komplexe Bestimmung der Länge des Bodies.**
4. **HTTP Response Messages als Bytestrome an Client senden.**
5. **Abbau der TCP-Verbindung durch den Client oder Server**
6. **Abbau der TCP-Verbindung durch den Server oder Client**



Was haben wir noch nicht untersucht?

- Wie lassen sich mit den HTTP-Methoden universelle APIs realisieren?
  - Wie werden komplexe Ressourcen, z.B. Datenbank adressiert?
  - Wie werden komplexe Operationen (Funktionsaufruf) aufgerufen?



- Wie kann Browser-Networking performant durchgeführt werden?
  - Wie wird TCP effizient genutzt?
  - Wie werden komplexe Datenstrukturen (z.B. komplexe Webseite) effizient geladen?
  - Wie erreicht man Parallelität?
  - Wie erreicht man Sicherheit
  - Datenkompression?

Geschichte von HTTP

1991: HTTP 0.9, Tim Berners-Lee (CERN) entwickelte HTTP.
Ziel: Einfachheit, einfach eine ASCII-Seite vom Server abrufen.

1992: HTTP 1.0, W3C.
Ziel: Übertragung beliebiger Dateiformate zwischen Client und Server.

1997: HTTP 1.1, RFC 2068.
Ziel: Performance-Optimierung, Caching, Pipelining.

2000: Roy Fielding: Doktorarbeit „Geburtsstunde der REST-API
(REST = REpresentational State Transfer)

2012: Erster HTTP 2.0 basierend auf SPDY (Google)

2015: HTTP 2.0. RFC 7540.
Konzept: Performance-Optimierung durch Multiplexing (parallele Anfragen), Stream-Priorisierung, Server-Push, „must“-TLS.

2018: Hypertext Transfer Protocol (HTTP) over QUIC
UDP-basiertes HTTP

2019: HTTP-over-QUIC geht über in HTTP/3.

Literaturhinweise

- IETF: *RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1*, https://datatracker.ietf.org/doc/html/rfc2616, June 1999.
- IETF: *RFC 7230 - Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*, https://datatracker.ietf.org/doc/html/rfc7230, June 2014.
- IETF: *RFC 7230 - Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*, https://datatracker.ietf.org/doc/html/rfc7231, June 2014.
- Mozilla and individual contributors: *HTTP*, https://developer.mozilla.org/en-US/docs/Web/HTTP (abgerufen am 13.05.2021)
- David Gourley, Brian Totty, Marjorie Sayer, Anshu Aggarwal, Sailu Reddy: *HTTP: The Definitive Guide*, O'Reilly Media, Inc., 2002