WBE: JAVASCRIPT

WEBSERVER

ÜBERSICHT

- Internet-Protokolle
- Das HTTP-Protokoll
- Node.js Webserver
- REST APIs
- Express.js

ÜBERSICHT

- Internet-Protokolle
- Das HTTP-Protokoll
- Node.js Webserver
- REST APIs
- Express.js

INTERNET

- Netzwerk von Internet-Geräten
- Internet-Protokoll-Stack (TCP/IP, ...)

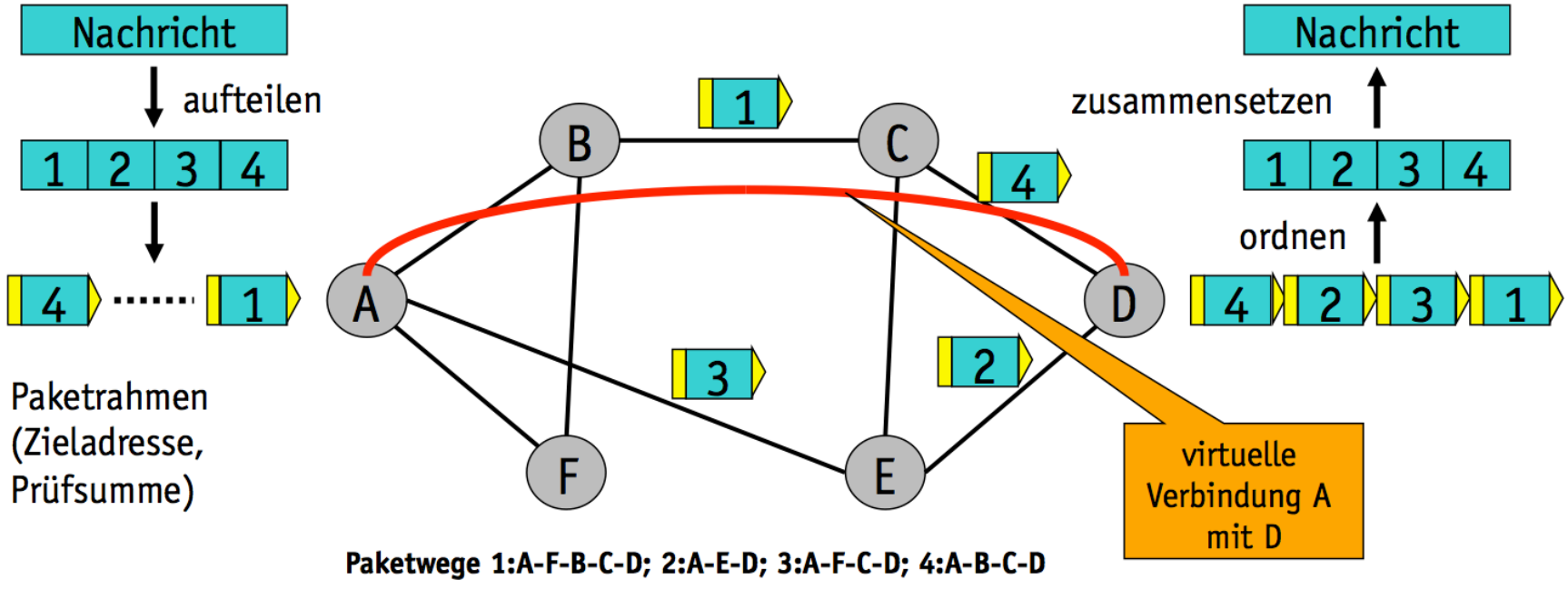
<table>
<tr><th>ISO/OSI</th><th>Internet</th><th>Protokolle</th><th>typische Angaben</th></tr>
<tr><td>Application (Anwendung)</td><td rowspan="2">Application</td><td rowspan="2">HTTP
FTP
SMTP
Telnet</td><td rowspan="2">URL:
http://www.zhwin.ch
Mailadresse:
mustepet@zhwin.ch</td></tr>
<tr><td>Presentation (Darstellung)</td></tr>
<tr><td rowspan="2">Session (Sitzung)</td><td rowspan="2">Transport</td><td rowspan="2">TCP
UDP</td><td rowspan="2">Portnummer
80 = HTTP
25 = SMTP</td></tr>
<tr><td>Transport</td></tr>
<tr><td>Network (Netzwerk)</td><td>Internet</td><td>IP</td><td>IP-Adresse
192.168.0.1</td></tr>
<tr><td>Data Link (Sicherung)</td><td rowspan="2">Physical / Access</td><td rowspan="2">Ethernet
Wireless LAN
Token Ring
PPP/(Modem, ISDN, xDSL)</td><td rowspan="2">MAC-Adresse
00:0F:7F:23:45:67
Telefonnummer:
0878/123456</td></tr>
<tr><td>Physical (Bitübertragung)</td></tr>
</table>

Speaker notes

HTTP ist ein Protokoll auf der Applikations-Ebene. Ebenso wie die Filetransfer- oder E-Mail-Protokolle. Die Transport-Schicht erlaubt es, mittels Ports verschiedene Applikationen auf einem Host zu adressieren, sowie, in Form von TCP, zuverlässige Verbindungen herzustellen. Das Internet-Protokoll vermittelt Pakete unabhängig vom Übertragungsmedium.

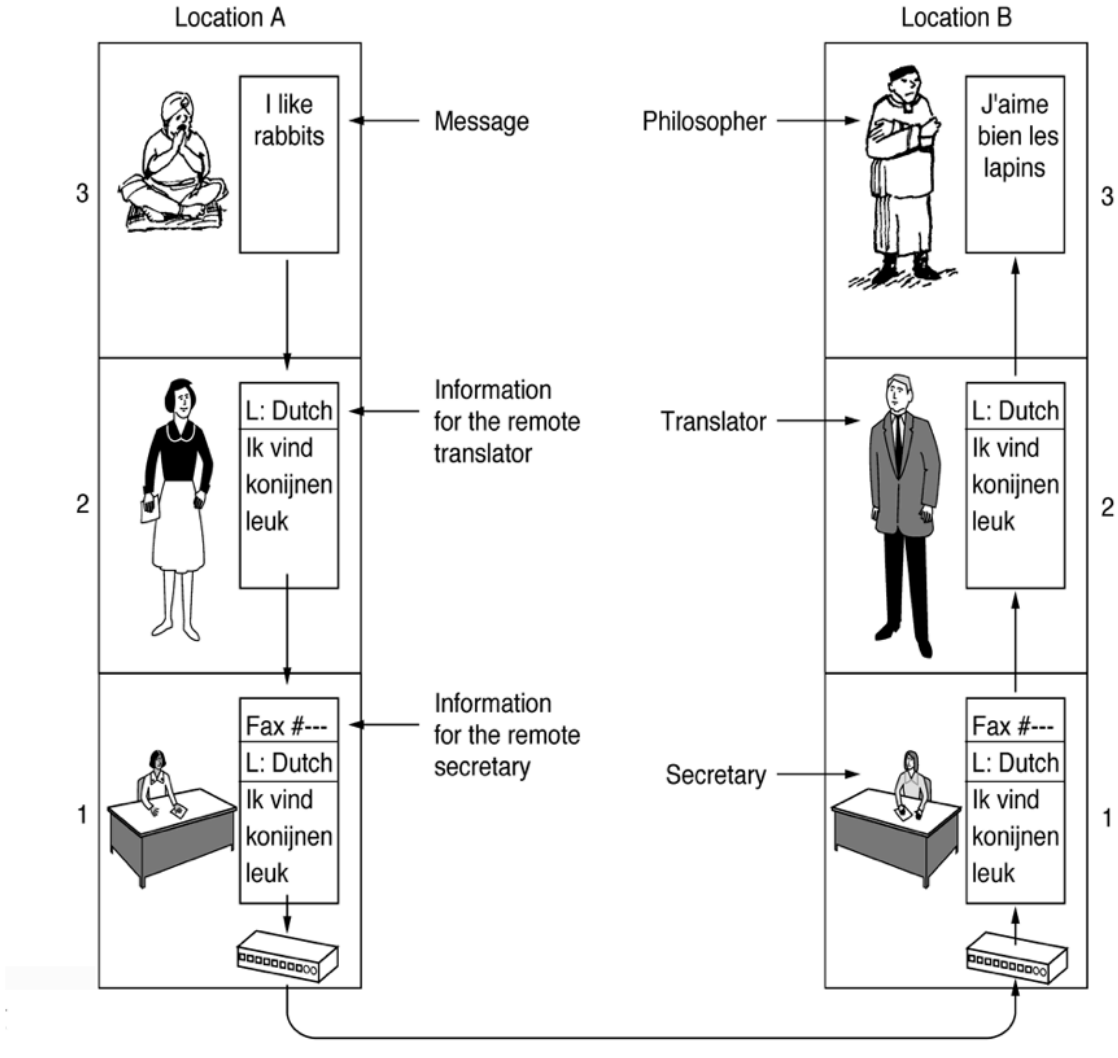
Mehr zu den Ebenen des Protokoll-Stacks erfahren Sie in anderen Fächern.

Prinzip der Paketvermittlung



Paketwege 1:A-F-B-C-D; 2:A-E-D; 3:A-F-C-D; 4:A-B-C-D

Veranschaulichung des Schichtenmodells



Quelle:
Andrew Tanenbaum and David Wetherall:
Computer Networks, Fifth Edition, Prentice Hall, 2011.

Eine etwas ausführlichere Beschreibung finden Sie auch im Skript zum WBE-Vorkurs.

Weitere Infos:

Wikipedia: Internet protocol suite
https://en.wikipedia.org/wiki/Internet_protocol_suite

Wikipedia: Internetprotokollfamilie
https://de.wikipedia.org/wiki/Internetprotokollfamilie

Die TCP/IP Protokollfamilie
https://www.linux-praxis.de/die-tcp-ip-protokollfamilie

DNS: The Good Parts
https://www.petekeen.net/dns-the-good-parts/

SERVER IM INTERNET

- Wartet auf Anfragen auf bestimmtem Port
- Client stellt Verbindung her und sendet Anfrage
- Server beantwortet Anfrage

Port	Service
20	FTP -- Data
21	FTP -- Control
22	SSH Remote Login Protocol
23	Telnet
25	Simple Mail Transfer Protocol (SMTP)
53	Domain Name System (DNS)
80	HTTP
443	HTTPS

WEBSERVER

- Wartet auf HTTP/S-Anfragen
- Standard-Ports: 80, 443
- Beispiele: Nginx, Apache Webserver, Apache Tomcat

```
GET /index.html HTTP/1.1
```

DAS WEB (WH)

- Client: Browser (oder allgemein: User Agent)
- Server: Web Server

- Protokoll: HTTP/S
- Sprachen: HTML, CSS, ...
- Adressierung: URL/URI

```
 http://eloquentjavascript.net/13_browser.html
|      |                        |               |
protocol       server                path
```

Speaker notes

HTTP-Adressen werden vom Browser übers Netz von einem Webserver angefordert. Wenn nur statische Dateien geladen werden sollen, können diese ohne Umweg über einen Server auch über eine file-Adresse geladen werden:

```
file:///my_demos/13_browser.html
```

Dies funktioniert aber nicht, sobald serverseitige Programmlogik involviert ist, etwa bei der Auswertung von Formulardaten. Ausserdem gibt es in modernen Browsern beim Laden vom Scripts teilweise Einschränkungen beim Laden vom Filesystem, auch wenn die zugehörigen HTML- und CSS-Dateien bereits vom Filesystem geladen wurden.

SERVER AN DER ZHAW

https://dublin.zhaw.ch/~<kurzzeichen>

- Laborserver: CGI, PHP, MySQL, Postgres
- Zugang nur noch innerhalb des ZHAW-Netzes (oder VPN)

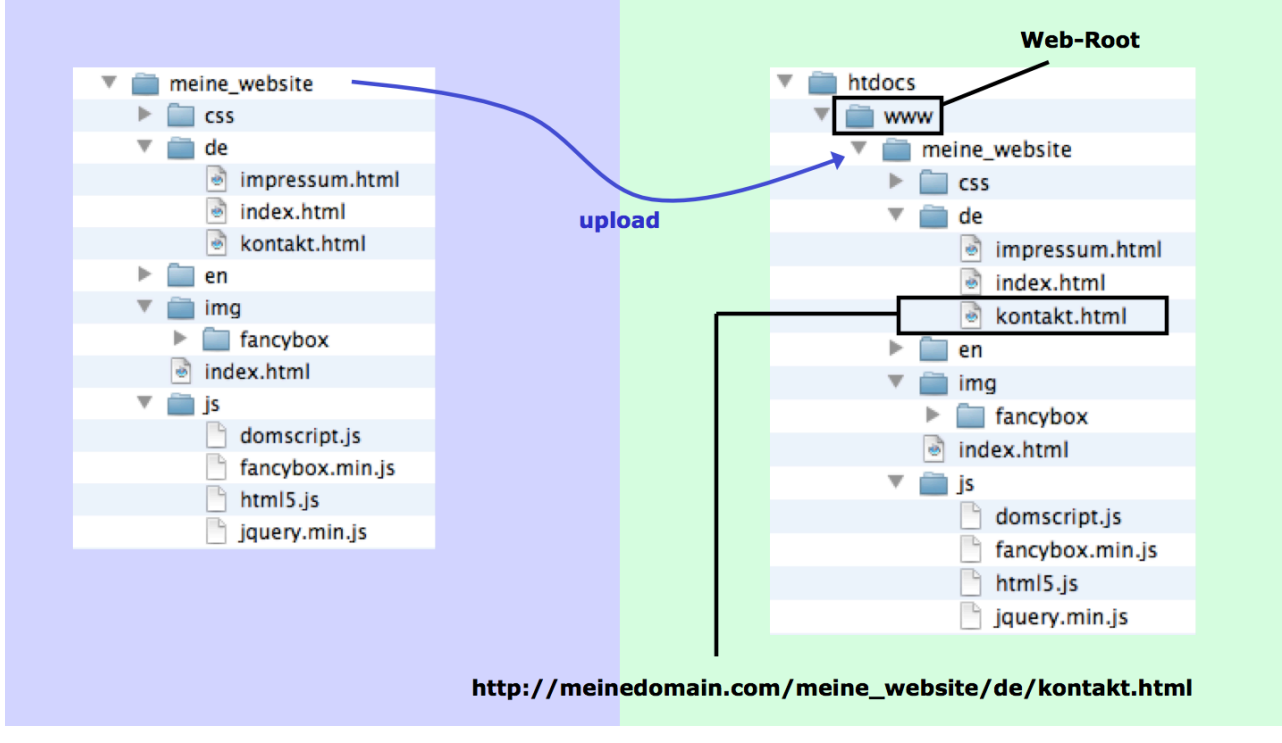
https://github.zhaw.ch/

- Github Pages

Zahlreiche weitere Labor- und Test-Server für bestimmte Aufgaben, teilweise von ausserhalb des ZHAW-Netzes erreichbar, teilweise nur über VPN

WEB-ROOT

- Einstellung des Web-Servers
- Stelle im Server-Verzeichnis, welche Wurzel des Web-Verzeichnisses ist



FILE-TRANSFER

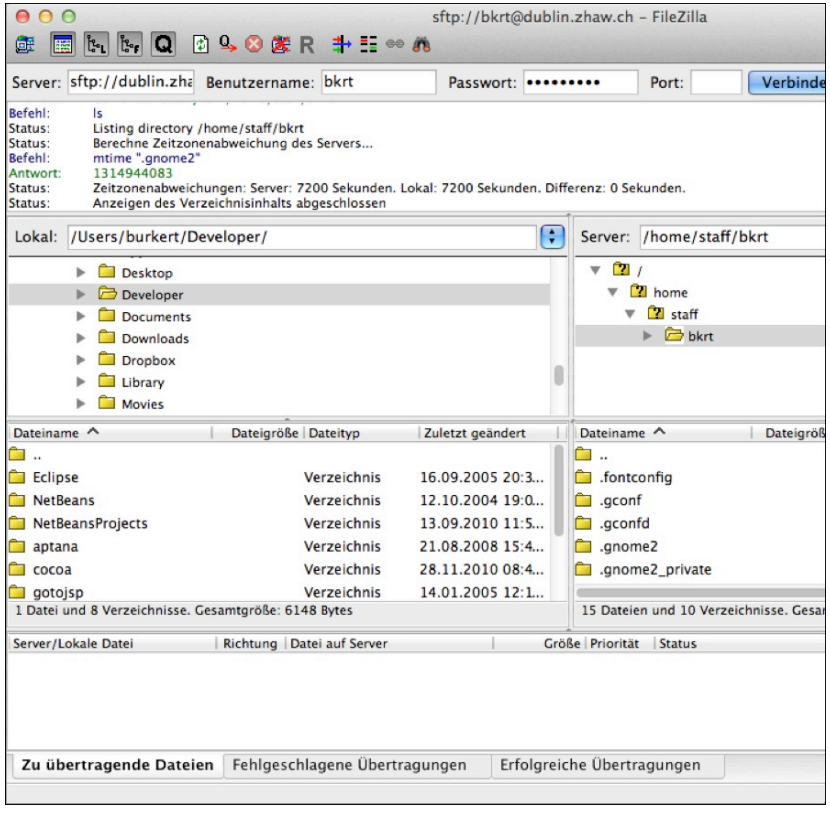
- FTP (File Transfer Protocol)
- SFTP (SSH File Transfer Protocol)
- Anwendungen mit GUI und auf der Kommandozeile (ftp, sftp)

```
$ sftp bkrt@dublin.zhaw.ch
password:
Connected to dublin.zhaw.ch.

sftp> dir
ggt.py      ggt.pyc     index.html  private     public      www

sftp> get ggt.py
Fetching /home/staff/bkrt/ggt.py to ggt.py
/home/staff/bkrt/ggt.py
100%   58     0.1KB/s   00:00

sftp> quit
$
```

Speaker notes

FileZilla

- Robuster S/FTP-Client
- Windows, Linux, Mac
- Open Source



https://filezilla-project.org

SECURE SHELL: SSH

- Sichere Verbindung zum Server herstellen
- Dort auf der Kommandozeile arbeiten

```
$ ssh dublin.zhaw.ch -l bkrt
bkrt@dublin.zhaw.ch's password:
Last login: Tue Jul 16 13:47:05 2013 from ...

$ ls
ggt.py  ggt.pyc  index.html  ine1  private  public  www

$ cd www
$ mv index.html old.html
$ exit
```

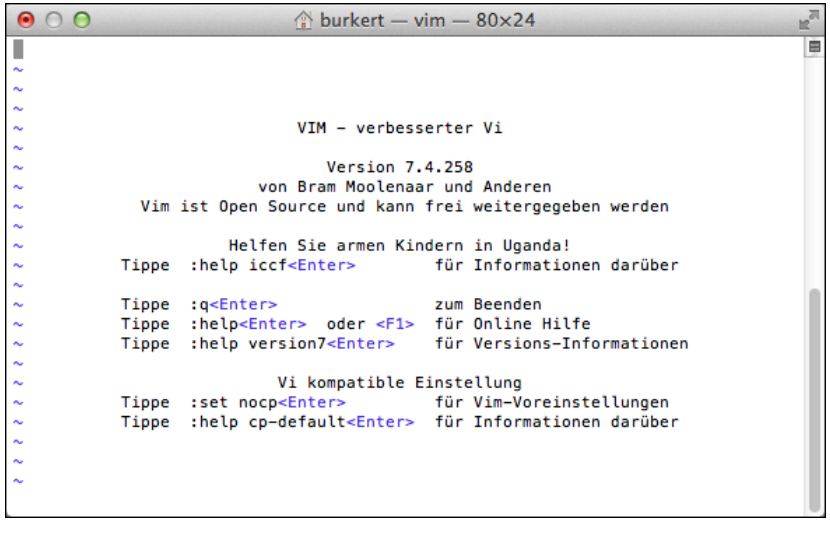
Speaker notes

Die wichtigsten Kommandos sollte man kennen...

- Dateien auflisten, verschieben und kopieren (ls, mv, cp)
- Berechtigungen und Eigentümer anpassen (chown, chmod)
- Prozesse und Festplattenauslastung ansehen (ps, top, df)
- Dateitransfer und Verbinden mit anderem Host (sftp, scp, ssh)
- Netzwerkbefehle (ping, traceroute, host, dig)
- Dateien bearbeiten (vi)

vi (vim)

- Gilt als komplizierter Editor
- Ein paar wichtige Befehle hat man sich aber schnell angeeignet



ÜBERSICHT

- Internet-Protokolle
- Das HTTP-Protokoll
- Node.js Webserver
- REST APIs
- Express.js

HTTP

Aufruf http://dublin.zhaw.ch/~bkrt/hallo.html im Browser

- DNS-Abfrage `dublin.zhaw.ch`
- Liefert IP-Adresse, z.B.: `160.85.67.138`
- Verbindung zu Host auf Port 80 herstellen
- HTTP-Anfrage senden: `GET /~bkrt/hallo.html HTTP/1.1`
- Server sendet Antwort und beendet Verbindung

HTTP REQUEST

```
GET /~bkrt/hallo.html HTTP/1.1
Host: dublin.zhaw.ch
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X...) Gecko/20100101 Firefox
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: de-de,de;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Speaker notes

Die erste Zeile ist hier wesentlich: hier steht, welche Resource wie angesprochen werden soll. Die Methode hier ist `GET`, d.h. man möchte die Ressource laden.

Der `Host`-Header ist nötig, damit der Server die Website zuordnen kann, wenn mehrere Websites mit verschiedenen Domainnamen auf dem Server gehostet werden.

Nach dem Header können HTTP-Requests noch einen Body mit den zu übermittelnden Daten haben. Dieser ist vom Header durch eine Leerzeile getrennt. GET-Requests haben keinen Body, andere Methoden teilweise schon, z.B. POST.

HTTP REQUEST: METHODEN

- `GET`: Ressource laden
- `POST`: Informationen senden
- `PUT`: Ressource anlegen, überschreiben
- `PATCH`: Ressource anpassen
- `DELETE`: Ressource löschen ...

https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_method

HTTP RESPONSE

```
HTTP/1.1 200 OK
Date: Mon, 15 Jul 2013 17:10:56 GMT
Server: Apache/2.2.15 (CentOS)
Last-Modified: Wed, 17 Oct 2012 08:10:22 GMT
ETag: "5b018a-af-4cc3ccd575780"
Accept-Ranges: bytes
Content-Length: 175
Connection: close
Content-Type: text/html; charset=UTF-8

<!doctype html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Hallo</title>
  </head>
  <body>
    <h1>Hallo</h1>
    <p>Ich bin eine Webseite</p>
  </body>
</html>
```

Speaker notes

HTTP-Header und -Body (hat nichts mit dem HTML-Element `<body>` zu tun) sind durch eine Leerzeile voneinander getrennt.

Der HTTP Header kann weitere Zeilen enthalten, jeweils in der Form `name: value`. Bei der Anfrage zum Beispiel Informationen zum Browser und dessen Features. Bei der Antwort Informationen zum Server und andere Angaben. Wichtig sind hier auch Grösse (Content-Length) und Typ (Content-Type) des Inhalts. Bei Textdateien kann der Content-Type auch noch die Zeichencodierung enthalten.

Die Angabe Content-Type entspricht übrigens den MIME-Types der E-Mails. MIME = Multipurpose Internet Mail Extensions. Damit sind verschiedene Formate und Attachments in E-Mails möglich.

Media Types
https://www.iana.org/assignments/media-types/media-types.xhtml

HTTP RESPONSE: STATUS CODES

- `1XX`: Information (z.B. 101 Switching Protocols)
- `2XX`: Erfolg (z.B. 200 Ok, 204 No Content)
- `3XX`: Weiterleitung (z.B. 301 Moved Permanently)
- `4XX`: Fehler in Anfrage (z.B. 403 Forbidden, 404 Not Found)
- `5XX`: Server-Fehler (z.B. 501 Not Implemented)

https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

ÜBERSICHT

- Internet-Protokolle
- Das HTTP-Protokoll
- Node.js Webserver
- REST APIs
- Express.js

EINFACHER WEBSERVER

```
const {createServer} = require("http")

let server = createServer((request, response) => {
  response.writeHead(200, {"Content-Type": "text/html"})
  response.write(`
    <h1>Hello!</h1>
    <p>You asked for <code>${request.url}</code></p>`)
  response.end()
})
server.listen(8000)
console.log("Listening! (port 8000)")
```

Speaker notes

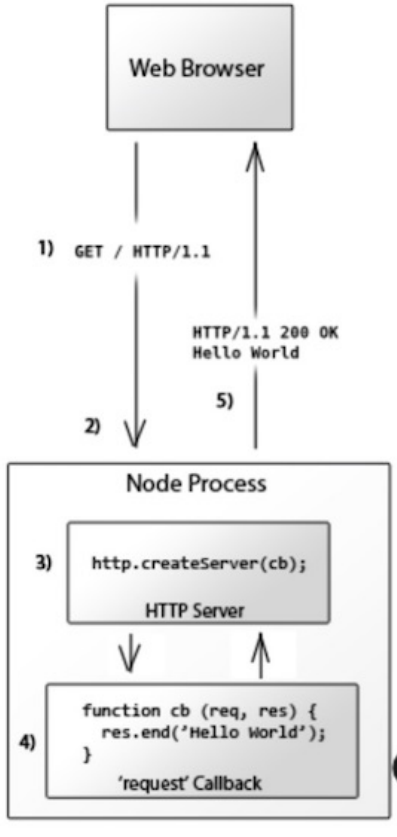
Aufruf zum Beispiel:
http://localhost:8000/hello

Das Callback wird bei jeder Verbindungsaufnahme zum Server aufgerufen. Die Parameter `request` und `response` repräsentieren eingehende und ausgehende Daten.

`response.write()` kann auch mehrfach aufgerufen werden, wenn Daten zum Client nach und nach übermittelt werden sollen, sobald sie verfügbar sind.

Nachdem das Script gestartet wurde, wartet es auf Anfragen. Es kann mit `CTRL-C` beendet werden.

EINFACHER WEBSERVER



1. An HTTP client, like a web browser initiates an HTTP request
2. Node accepts the connection and incoming request data is given to the HTTP Server
3. HTTP Server parses up to the end of the HTTP headers, and then hands logic over to the 'request' callback
4. Request callback performs application logic, in this case responding immediately with the text "Hello World"
5. The response is sent back through the HTTP Server, which formats a proper HTTP response to send back to the client

Speaker notes

Dem Beispiel fehlen noch viele Funktionen typischer Webserver:

- Routing basierend auf dem Pfad der URL
- Sessions (z.B. mit Cookies)
- Verarbeiten eingehender Daten (Formulardaten, JSON)
- Zurückweisen fehlerhafter Anfragen

EINFACHER WEB-CLIENT

```
const {request} = require("http")

let requestStream = request({
  hostname: "eloquentjavascript.net",
  path: "/20_node.html",
  method: "GET",
  headers: {Accept: "text/html"}
}, response => {
  console.log("Server responded with status code", response.statusCode)
})

requestStream.end()
```

Speaker notes

Hier wird die `request`-Funktion des http-Moduls verwendet. Das erste Argument dient zur Konfiguration der Anfrage. Das zweite Argument ist die Callback-Funktion, welche mit dem `response`-Objekt aufgerufen wird.

Bei der GET-Anfrage im Beispiel wird kein HTTP-Body übermittelt. Ansonsten könnten vor dem `requestStream.end()` noch `requestStream.write()`-Aufrufe eingefügt werden.

NODE.JS WEB-CLIENT

- Einfache Variante mit `http`-Modul (letztes Beispiel)
- Paket `https` für HTTPS-Zugriffe
- Seit Node.js 18 wird auch die Fetch API unterstützt (mehr dazu beim Thema „Client-Server-Interaktion”)
- Alternative: Axios, HTTP-Client für Browser und Node.js

Speaker notes

Beispiel für GET-Request mit Axios:

```
const axios = require('axios')

// Make a request for a user with a given ID
axios.get('/user?ID=12345')
  .then(function (response) {
    // handle success
    console.log(response)
  })
  .catch(function (error) {
    // handle error
    console.log(error)
  })
  .finally(function () {
    // always executed
  })
```

Beispiel für POST-Request:

```
axios.post('/user', {
  firstName: 'Fred',
  lastName: 'Flintstone'
})
.then(function (response) {
  console.log(response)
})
.catch(function (error) {
  console.log(error)
})
```

https://www.npmjs.com/package/axios

STREAMS: SERVER

```
const {createServer} = require("http")

createServer((request, response) => {
  response.writeHead(200, {"Content-Type": "text/plain"})
  request.on("data", chunk =>
    response.write(chunk.toString().toUpperCase()))
  request.on("end", () => response.end())
}).listen(8000)
```

- Eingehende Daten als Stream gelesen
- `data`-Event: nächster Teil verfügbar
- `end`-Event: alle Daten wurden übertragen

Speaker notes

Server, der eingehende Daten nach Grossbuchstaben umwandelt und direkt wieder auf den Ausgabe-Stream schreibt.

STREAMS: CLIENT

```
const {request} = require("http")

let rq = request({
  hostname: "localhost",
  port: 8000,
  method: "POST"
}, response => {
  response.on("data", chunk =>
    process.stdout.write(chunk.toString()));
})

rq.write("Hello server\n")
rq.write("And good bye\n")
rq.end()
```

Speaker notes

Ausgabe:

```
HELLO SERVER
AND GOOD BYE
```

Die Ausgabe erfolgt hier mit `process.stdout.write`. Im Gegensatz zu `console.log` wird hier nicht automatisch ein `\n` angehängt.

BEISPIEL: FILE-SERVER (1)

(Exkurs)

- Kleiner Server zum Zugriff auf Files
- HTTP-Methoden `GET`, `DELETE` und `PUT`
- Im Unterricht nur kurzer Überblick über Funktionsweise
- Kompletter Code in Demos, Erklärungen in Lecture Notes
- Beispiel, wie bestimmte Features umgesetzt werden können
- Achtung: nicht für produktiven Einsatz im Web geeignet

Speaker notes

Das Beispiel auf den folgenden Slides zeigt, wie mit überschaubarem Aufwand ein Server mit ein paar interessanten Funktionen aufgebaut werden kann. Achtung: Es ist ganz und gar keine gute Idee, einen solchen Server offen ins Internet zu stellen, da nur wenige Sicherheits-Features eingebaut sind und jede Art der Authentifizierung fehlt.

Noch ein Hinweis: Es hat ziemlich viel Code auf dieser und den folgenden Slides. Im Unterricht wird relativ schnell über dieses Beispiel gegangen, da es mühsam wäre, jedes Detail im Unterricht zu besprechen. Sehen Sie dieses Beispiel daher als kleinen Exkurs an mit ein paar Anregungen, wie bestimmte Features umgesetzt werden können.

Quelle:
https://eloquentjavascript.net/20_node.html#h_yAdw1Y7bgN

BEISPIEL: FILE-SERVER (2)

```
1  const {createServer} = require("http")
2  const methods = Object.create(null)
3
4  createServer((request, response) => {
5    let handler = methods[request.method] || notAllowed;
6    handler(request)
7      .catch(error => {
8        if (error.status != null) return error
9        return { body: String(error), status: 500 }
10     })
11     .then(({body, status=200, type="text/plain"}) => {
12       response.writeHead(status, {"Content-Type": type})
13       if (body && body.pipe) body.pipe(response)
14       else response.end(body)
15     })
16 }).listen(8000)
```

Speaker notes

- Eigener Handler für jede HTTP-Methode (GET, PUT, ...)
- Zunächst wird Handler ausgewählt
- Falls keiner oder kein passender: notAllowed
- Dann wird Handler mit dem Request aufgerufen
- Liefert eine Promise zurück
- Falls *rejected*: Fehlerantwort angelegt
- Falls Stream: mit pipe auf *response* umleiten
- Sonst: body als Inhalt senden

Die Parameterliste im `then`-Callback ist noch interessant:

```
({body, status=200, type="text/plain"}) => ...
```

Übergeben wird hier ein Objekt. Dieses wird destrukturiert, d.h. die Attribute *body*, *status* und *type* werden herausgepflückt. Zusätzlich gibt es Defaults für die Attribute, für den Fall, dass das übergebene Objekt sie nicht enthält. Wir haben hier also: „object destructuring with defaults”.

BEISPIEL: FILE-SERVER (3)

```
async function notAllowed (request) {
  return {
    status: 405,
    body: `Method ${request.method} not allowed.`
  }
}
```

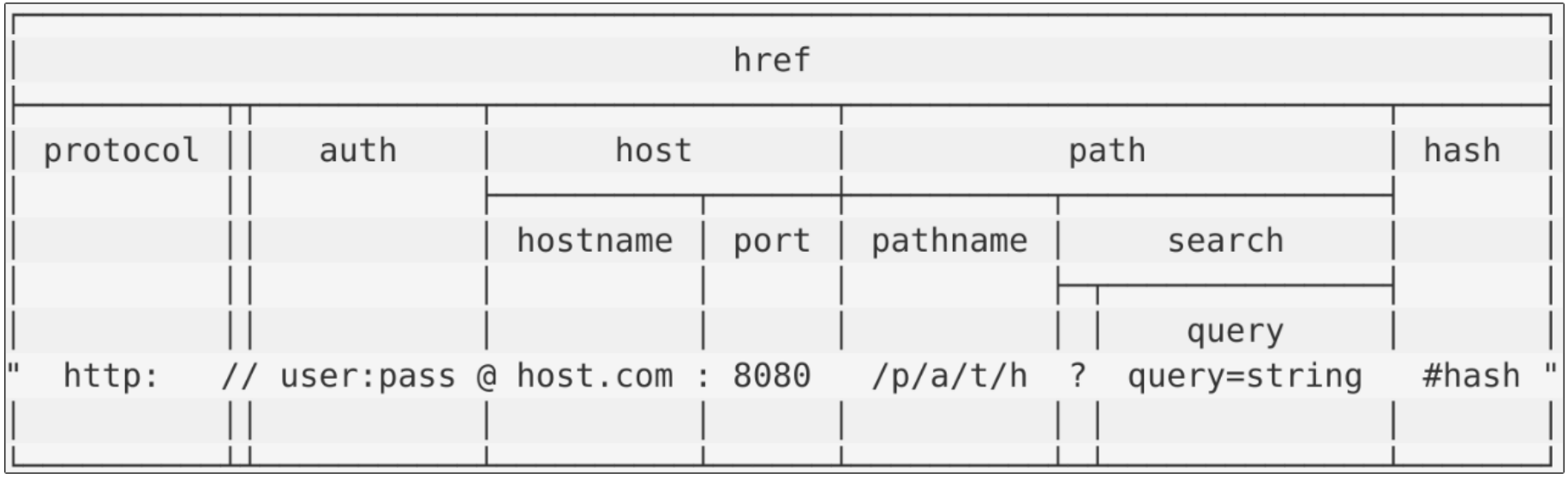
- Unbekannter Handler
- `notAllowed` (405) senden

BEISPIEL: FILE-SERVER (4)

```
1  const {parse} = require("url")
2  const {resolve, sep} = require("path")
3
4  const baseDirectory = process.cwd()
5
6  function urlPath (url) {
7    let {pathname} = parse(url)
8    let path = resolve(decodeURIComponent(pathname).slice(1))
9    if (path != baseDirectory && !path.startsWith(baseDirectory + sep)) {
10     throw {status: 403, body: "Forbidden"}
11   }
12   return path
13 }
```

Speaker notes

Hier wird der Pfad analysiert und relative Pfadangaben (können zu, Beispiel ".." enthalten, potentielles Sicherheitsrisiko!) ausgewertet. Dazu dient die Funktion `resolve`. Erlaubt werden schliesslich nur Pfadangaben, die dem Ausgangsverzeichnis entsprechen oder mit diesem beginnen.



BEISPIEL: FILE-SERVER (5)

```
const {createReadStream} = require("fs")
const {stat, readdir} = require("fs").promises
const mime = require("mime")

methods.GET = async function (request) {
  let path = urlPath(request.url)
  let stats
  try {
    stats = await stat(path)
  } catch (error) {
    if (error.code != "ENOENT") throw error
    else return {status: 404, body: "File not found"}
  }
  if (stats.isDirectory()) {
    return {body: (await readdir(path)).join("\n")}
  } else {
    return {body: createReadStream(path),
            type: mime.getType(path)}
  }
}
```

Speaker notes

Das ist die GET-Methode. Hier wird das Promise-basierende FileSystem-API verwendet. Das `mime`-Paket ermittelt den passenden ContentType zu der Datei. Falls es sich um ein Verzeichnis handelt, wird das Array der Einträge in einen String (ein Eintrag pro Zeile) umgewandelt und mit dem Default-ContentType *text/plain* zurückgeliefert.

`EN0ENT` steht für "Error NO ENTry". Es ist somit eine Abkürzung, weil C-Compiler vor vielen Jahren (Jahrzehnten) nur 8 Zeichen in Symbolen erlaubten.

BEISPIEL: FILE-SERVER (6)

```
1  const {rmdir, unlink} = require("fs").promises
2
3  methods.DELETE = async function (request) {
4    let path = urlPath(request.url)
5    let stats
6    try {
7      stats = await stat(path)
8    } catch (error) {
9      if (error.code != "ENOENT") throw error
10     else return {status: 204}
11   }
12   if (stats.isDirectory()) await rmdir(path)
13   else await unlink(path)
14   return {status: 204}
15 }
```

Speaker notes

Falls die zu löschende Datei nicht vorhanden ist, wird ein 204 (“no content”) zurückgegeben. Eine nicht vorhandene Datei zu löschen ist also immer erfolgreich: Das Ziel (Datei nicht vorhanden) wird immer erreicht. Ansonsten wird je nach Inhalt `rmdir` oder `unlink` zum Löschen verwendet.

BEISPIEL: FILE-SERVER (7)

```
1  const {createWriteStream} = require("fs");
2
3  function pipeStream (from, to) {
4    return new Promise((resolve, reject) => {
5      from.on("error", reject)
6      to.on("error", reject)
7      to.on("finish", resolve)
8      from.pipe(to)
9    })
10 }
11
12 methods.PUT = async function (request) {
13   let path = urlPath(request.url)
14   await pipeStream(request, createWriteStream(path))
15   return {status: 204}
16 }
```

Speaker notes

Schliesslich noch PUT zum Hochladen einer Datei. Der `request`-Stream wird mit `pipe` zu einem FileStream umgeleitet, der mit dem gewünschten Pfad angelegt wurde. Da `pipe` kein Promise-API hat, wird es in eine Promise gekapselt.

BEISPIEL: FILE-SERVER (8)

Test des Servers:

```
$ curl http://localhost:8000/file.txt
File not found
$ curl -X PUT -d hello http://localhost:8000/file.txt
$ curl http://localhost:8000/file.txt
hello
$ curl -X DELETE http://localhost:8000/file.txt
$ curl http://localhost:8000/file.txt
File not found
```

https://eloquentjavascript.net/20_node.html#h_yAdw1Y7bgN

Speaker notes

`curl` ist ein Kommandozeilen-Tool, um (u.a.) HTTP-Anfragen an Server machen zu können (https://curl.haxx.se).
Optionen:

- `-i`: HTTP-Headers ebenfalls mit anzeigen
- `-X [method]`: HTTP-Methode, zum Beispiel `-X POST`
- `-H [header]`: Header setzen, z.B. `-H "Content-Type: application/json"`
- `-d [json]`: HTTP-POST-Daten, z.B. `-d '{"username":"xyz","password":"abc"}'`

ÜBERSICHT

- Internet-Protokolle
- Das HTTP-Protokoll
- Node.js Webserver
- REST APIs
- Express.js

REST APIS

- REST: Representational State Transfer
- Programmierparadigma für verteilte Systeme
- Grundlage: Web-Architektur und HTTP
- Leichtgewichtig (im Vergleich zu RPC oder SOAP/WSDL)

Speaker notes

RPC steht für *Remote Procedure Calls*, SOAP für *Simple Object Access Protocol*, WSDL für *Web Services Description Language*. Das sind Sprachen und Protokolle für auf XML basierende Web-Services, welche nicht zum Stoff von WBE gehören.

In WBE wird eine kurze Einführung ins Thema REST gegeben, obwohl zu diesem Thema noch sehr viel mehr zu sagen wäre. Hier soll nur noch auf einen Artikel von Martin Fowler hingewiesen werden:

Richardson Maturity Model, steps toward the glory of REST
https://martinfowler.com/articles/richardsonMaturityModel.html

REST EIGENSCHAFTEN

- Zugriff auf Ressourcen über ihre Adresse (URI)
- Kein Zustand: jede Anfrage komplett unabhängig
- Kein Bezug zu vorhergehenden Anfragen
- Alle nötigen Informationen in Anfrage enthalten
- Verwenden der HTTP-Methoden: `GET`, `PUT`, `POST`, ...

Speaker notes

Ressourcen können Verschiedenes sein, z.B.:

- A specific software release
- First weblog entry for November 11, 2013
- A road map for Switzerland
- Some information about dogs
- The next prime number after 1024
- A relationship between two acquaintances, Alice and Bob
- A list of open issues in the bug database

(Quelle: Slides von WEB2)

RESTFUL APIS

- Basisadresse, z.B. http://example.com/api/
- Sammlung von Ressourcen, z.B. http://example.com/api/products/
- Einzelne Ressource, z.B. http://example.com/api/products/17
- Medientyp für Ressource/n, z.B. JSON
- Zulässige Operationen, z.B. GET, PUT, POST, or DELETE

HTTP-METHODEN IN RESTFUL APIS

HTTP-Methode	Sammlung (Collection)	Einzel-Ressource
GET	Repräsentation für die Collection laden	Repräsentation für die Ressource laden
POST	Ressource unterhalb der angegebenen anlegen	Ressource in der angegebenen anlegen
PUT	Sammlung ersetzen oder anlegen	Ressource ersetzen oder anlegen
DELETE	Löscht die angegebene Sammlung	Löscht die angegebene Ressource
PATCH	Sammlung anpassen oder anlegen	Ressource anpassen oder anlegen

Speaker notes

Nicht ganz RESTful gemäss dieser Tabelle, aber REST-alike ist diese API (nicht mehr online):

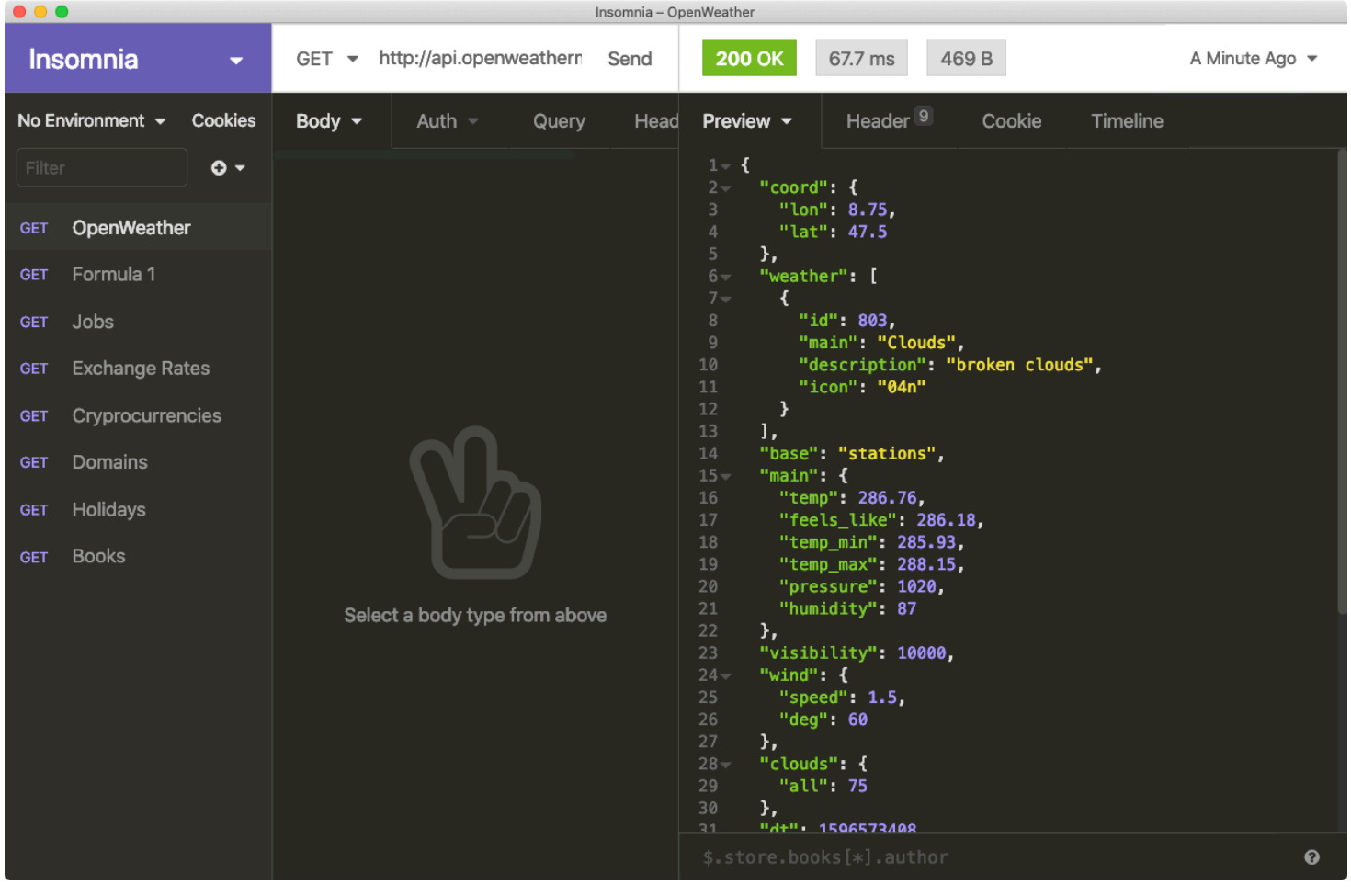
```
$ curl http://zhaw.herokuapp.com/task_lists/demo
{"id":"demo","title":"Demo Tasklist","tasks":[{"title":"Buy milk","done":false},

$ curl -d'{"tasks":[{"title": "Do homework"}]}' http://zhaw.herokuapp.com/task_l
{"tasks":[{"title":"Do homework"}],"id":"1596573186433604"}
```

REST APIS

- Viele Services stellen REST-APIs zur Verfügung
- In der Regel natürlich nur GET-Requests
- Beispiel: OpenWeather (Registrierung erforderlich)

```
$ curl "http://api.openweathermap.org/data/2.5/weather?q=Winterthur,ch&appid=674..."
{"coord":{"lon":8.75,"lat":47.5},"weather":[{"id":803,"main":"Clouds","description":
"broken clouds","icon":"04n"}],"base":"stations","main":{"temp":286.76,
"feels_like":286.18, "temp_min":285.93,"temp_max":288.15,"pressure":1020,...}}
```

REST TOOLS



https://insomnia.rest

REST-ALTERNATIVE: GRAPHQL

```
{
  hero {
    name
    friends {
      name
    }
  }
}
```

- Neueres Konzept, Facebook 2015
- Anfragesprache mit mächtigeren Auswahlmöglichkeiten
- Reihe von Werkzeugen zu diesem Zweck
- Im Beispiel: liefere alle `hero`-Einträge mit `name` und `friends`, von diesen aber auch nur `name`

Speaker notes

Beispielantwort:

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "friends": [
        {
          "name": "Luke Skywalker"
        },
        {
          "name": "Han Solo"
        },
        {
          "name": "Leia Organa"
        }
      ]
    }
  }
```

ÜBERSICHT

- Internet-Protokolle
- Das HTTP-Protokoll
- Node.js Webserver
- REST APIs
- Express.js

EXPRESS.JS

- Minimales, flexibles Framework für Web-Apps
- Zahlreiche Utilities und Erweiterungen
- Grundlage: Node.js
- Grundlage für zahlreiche weitere Frameworks

http://expressjs.com

Speaker notes

Ein Kandidat für die Nachfolge von Express ist koa:

koa – next generation web framework for node.js
https://koajs.com

INSTALLATION

```
$ mkdir myapp
$ cd myapp
$ npm init
$ npm install express --save
```

- Der Schritt `npm init` fragt eine Reihe von Informationen (Projektname, Version, ...) zum Projekt ab
- Als *Entry Point* ist hier `index.js` voreingestellt
- Das kann zum Beispiel in `app.js` geändert werden.

HELLO WORLD

```
1  const express = require('express')
2  const app = express()
3  const port = 3000
4
5  app.get('/', (req, res) => {
6    res.send('Hello World!')
7  })
8
9  app.listen(port, () => {
10   console.log(`Example app listening at http://localhost:${port}`)
11 })
```

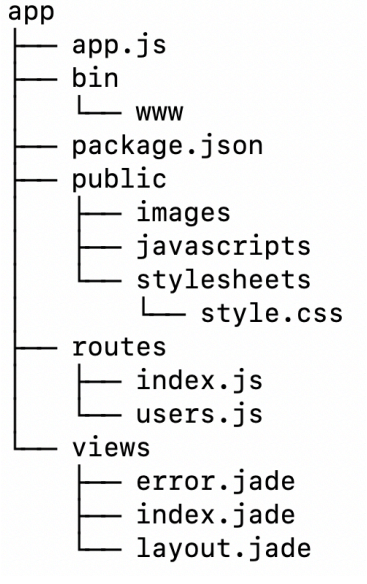
Speaker notes

Die `req`- und `res`-Objekte entsprechen denen in Node.js. Das heisst Aufrufe wie `req.pipe()` und `req.on('data', callback)` sind ebenfalls möglich.

Die Applikation antwortet in dieser Form auf Aufrufe von `http://localhost:3000/`, nicht aber auf `http://localhost:3000/hello`.

EXPRESS APP GENERATOR

- App-Gerüst mit häufig benötigten Komponenten anlegen
- Schnelle Variante zum Projektstart




```
# Hilfetext ausgeben
npx express-generator -h

# Generator starten
npx express-generator
```

http://expressjs.com/en/starter/generator.html

ROUTING

```
1  app.get('/', function (req, res) {
2    res.send('Hello World!')
3  })
4  app.post('/', function (req, res) {
5    res.send('Got a POST request')
6  })
7  app.put('/user', function (req, res) {
8    res.send('Got a PUT request at /user')
9  })
10 app.delete('/user', function (req, res) {
11   res.send('Got a DELETE request at /user')
12 })
```

http://expressjs.com/en/guide/routing.html

STATISCHE DATEIEN

- Middleware `express.static`
- Pfadangabe für Dateien als erstes Argument

```
1 app.use(express.static('public'))
2 /* http://localhost:3000/css/style.css
3 /* Pfad zur Datei: public/css/style.css
4 */
5 app.use('/static', express.static('public'))
6 /* http://localhost:3000/static/css/style.css
7 /* Pfad zur Datei: public/css/style.css
8 */
```

Speaker notes

Auch mehrere solche Angaben sind möglich. Die Verzeichnisse werden dann in der Reihenfolge der Angabe abgesucht.

MIDDLEWARE

- Funktionen mit Zugriff auf `request` und `response`
- Express-App ist eigentlich eine Folge von Middleware-Aufrufen

```
1 app.use(function (req, res, next) {
2   console.log('Time:', Date.now())
3   next()
4 })
5
6 app.use('/user/:id', function (req, res, next) {
7   console.log('Request Type:', req.method)
8   next()
9 })
```

Speaker notes

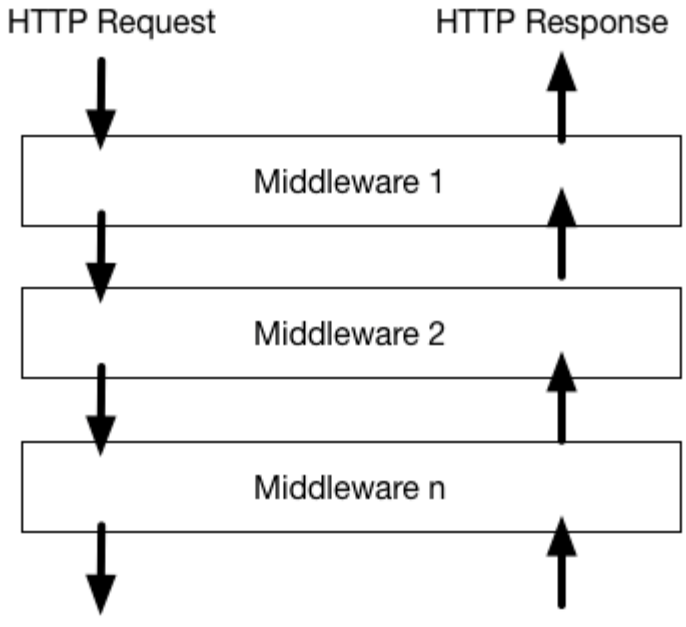
http://expressjs.com/en/guide/using-middleware.html

Weiteres Beispiel: Logging

```
const
  express = require('express'),
  morgan = require('morgan'),
  app = express()
app.use(morgan('dev'))
app.get('/api/:name', function(req, res) {
  res.status(200).json({'hello': req.params.name})
})
app.listen(8080, function(){
  console.log("ready captain.")
})
```

Morgan ist eine Middleware, um Zugriffe zur protokollieren. Im Beispiel werden alle Zugriffe im Entwicklermodus auf der Konsole protokolliert.

In `app.get('/api/:name', ...)` ist `:name` ein benannter Route-Parameter. Der definitive Wert wird aus dem Pfad entnommen und ist dann in `req.params` verfügbar.




```
response.writeHead(200);
response.write(file, 'binary‘);
response.end();
```

MIDDLEWARE

Module	Description
body-parser	Parse HTTP request body
compression	Compress HTTP responses
cookie-parser	Parse cookie header and populate req.cookies
cors	Enable cross-origin resource sharing (CORS)
passport	Authentication using “strategies” such as OAuth
…	…

http://expressjs.com/en/resources/middleware.html
http://www.passportjs.org

SERVER MIT NPM STARTEN

Eintrag in `package.json`:

```
"scripts": {
  "start": "node ./express_server.js"
}
```

Der Server kann dann so gestartet werden:

```
$ npm start
```

Speaker notes

Man kann auch einen `test`-Eintrag in die `scripts` einfügen, um einfach Tests mit npm anstossen zu können.

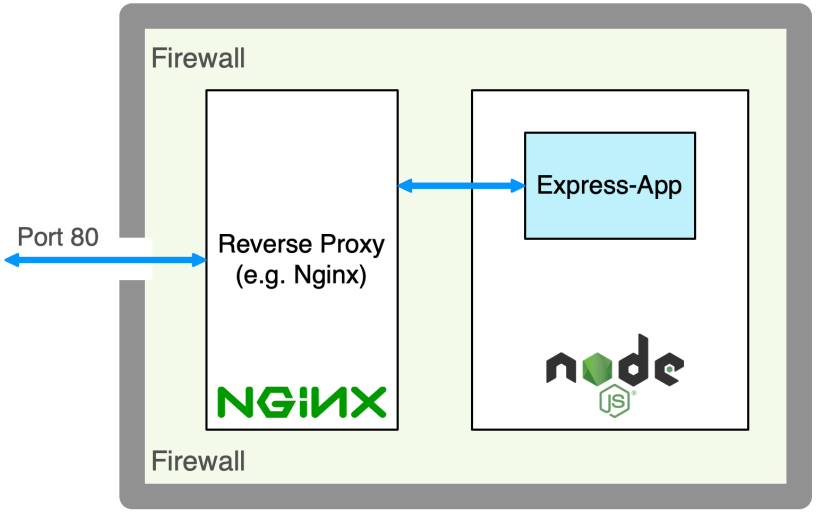
Noch eine Variante:

Mit dem Paket `nodemon` kann man einen Server starten, der nach jeder Änderung im Verzeichnis automatisch neu gestartet wird.

```
$ npm install -g nodemon
$ nodemon express_server.js
```

https://www.npmjs.com/package/nodemon

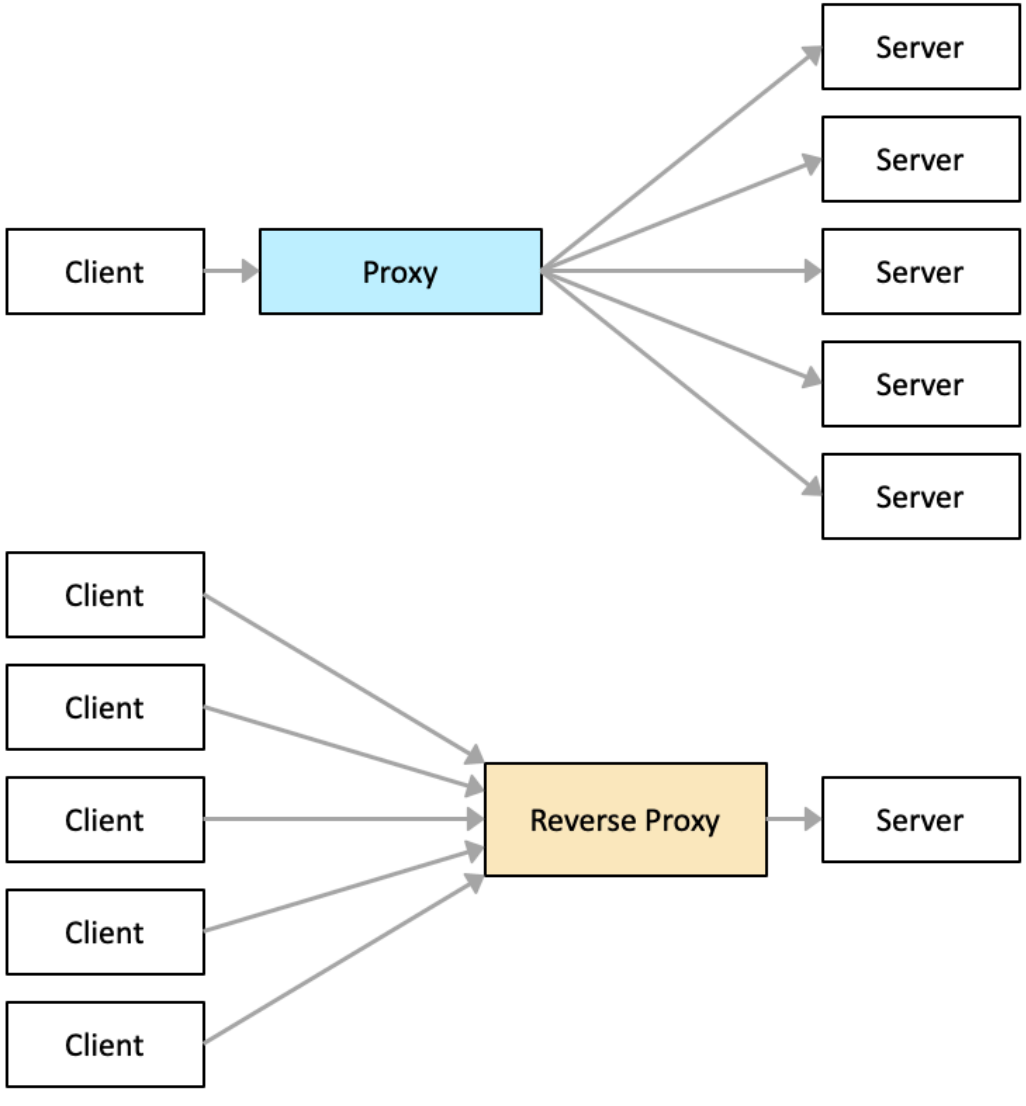
REVERSE PROXY



- Express-App wird übers Internet nicht direkt angesprochen
- Zugang erfolgt über Reverse Proxy, z.B. ein nginx-Server

- Dieser leitet Anfragen an die Express-App weiter
- Zusätzliche Services: Fehlerseiten, Komprimierung, Cache

Speaker notes



QUELLEN

- Marijn Haverbeke: Eloquent JavaScript
  https://eloquentjavascript.net/
- Ältere Slides aus WEB2 und WEB3
- Dokumentationen, u.a. zu Node.js

LESESTOFF

Geeignet zur Ergänzung und Vertiefung

- Einzelne Abschnitte in Kapitel 20 von:
  Marijn Haverbeke: Eloquent JavaScript
  https://eloquentjavascript.net/