WBE: BROWSER-TECHNOLOGIEN

CLIENT-SERVER-INTERAKTION

ÜBERSICHT

- Formulare
- Cookies, Sessions
- Ajax und XMLHttpRequest
- Fetch API

ÜBERSICHT

- Formulare
- Cookies, Sessions
- Ajax und XMLHttpRequest
- Fetch API

WARUM FORMULARE?

- Ermöglichen Benutzereingaben
- Grundlage für Interaktion im Web

```
<form>
  <input type="text" id="name">
  <input type="text" id="age">
  <input type="submit" value="Send">
</form>
```

FORMULARE: form

```
<form action="/login" method="post">
  ...
</form>
```

- Attribut `action`: Script das die Daten entgegen nimmt
- Attribut `method`: HTTP-Methode zum Senden der Daten
  - `GET`: Formulardaten an URL angehängt (Default)
  - `POST`: Formulardaten im Body des HTTP-Request gesendet

Beispiel: GET-Request
http://www.google.com/search?client=safari&q=http

Speaker notes

Attribut `action`:

- URL/Pfad zu einem Script, das die Eingaben verarbeitet
- Kann fehlen, wenn die Daten clientseitig verarbeitet werden

```
<form action="/login" method="post"> ... </form>
<form action="scripts/register.php" method="post"> ... </form>
<form action="search.cgi" method="get"> ... </form>
<form action="demo_form.asp" method="get"> ... </form>
```

Notes on GET:

- Appends form-data into the URL in name/value pairs
- The length of a URL is limited (about 3000 characters)
- Never use GET to send sensitive data! (will be visible in the URL)
- Useful for form submissions where a user wants to bookmark the result
- GET is better for non-secure data, like query strings in Google

Notes on POST:

- Appends form-data inside the body of the HTTP request (data is not shown in URL)
- Has no size limitations
- Form submissions with POST cannot be bookmarked

Source: https://www.w3schools.com/tags/att_form_method.asp

BESCHREIBUNG: label

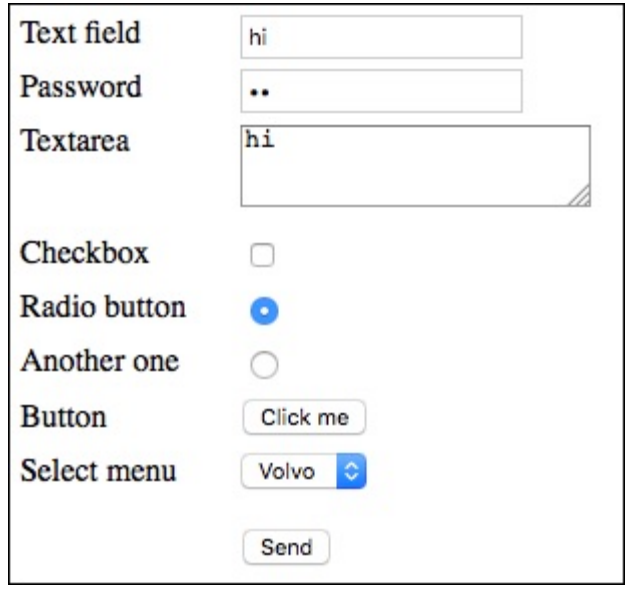
```
<form>
  <label>Name: <input type="text"></label>
  <label>Age: <input type="text"></label>
  <input type="submit" value="Send">
</form>

<form>
  <label for="nameid">Name: </label>
  <input type="text" id="nameid">
  <label for="ageid">Age: </label>
  <input type="text" id="ageid">
  <input type="submit" value="Send">
</form>
```

Name: Age: Send

- Das label-Element *beschreibt* ein Formularelement
- Klick auf Label setzt den Fokus auf das Element

FORMULARELEMENTE

```
<form>
  <label>Text field <input type="text" value="hi"></label>
  <label>Password <input type="password" value="hi"></label>
  <label>Textarea <textarea>hi</textarea></label>
  <label>Checkbox <input type="checkbox"></label>
  <label>Radio button <input type="radio" name="demo" checked></label>
  <label>Another one <input type="radio" name="demo"></label>
  <label>Button <button>Click me</button></label>
  <label>Select box
    <select name="cars">
      <option value="volvo">Volvo</option>
      <option value="saab">Saab</option>
      <option value="fiat">Fiat</option>
      <option value="audi">Audi</option>
    </select>
  </label>
  <input type="submit" value="Send">
</form>
```

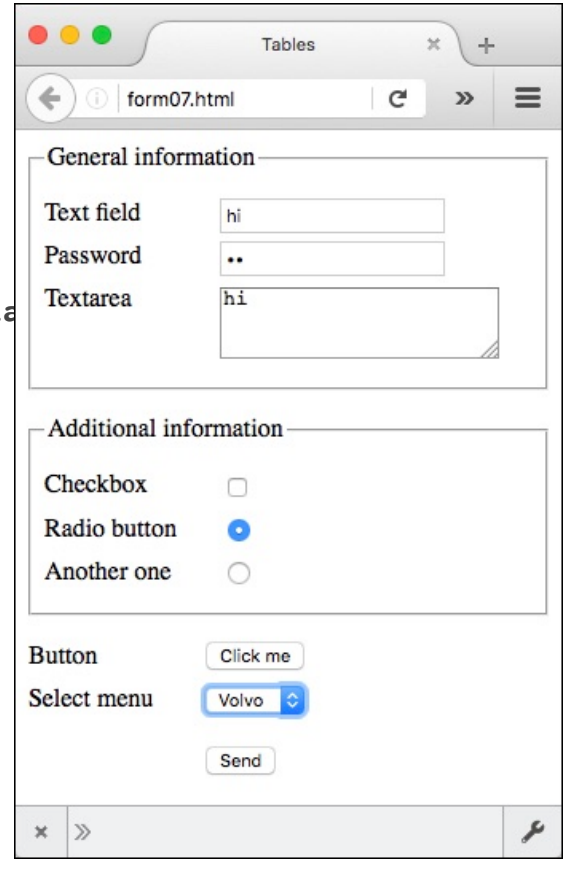


Speaker notes

Die wichtigsten Formularelemente sollten Sie kennen. Seltener benötigte wie Auswahlboxen und spezielle Attribute kann man bei Bedarf nachschlagen.

GRUPPIEREN VON FORMULARELEMENTEN

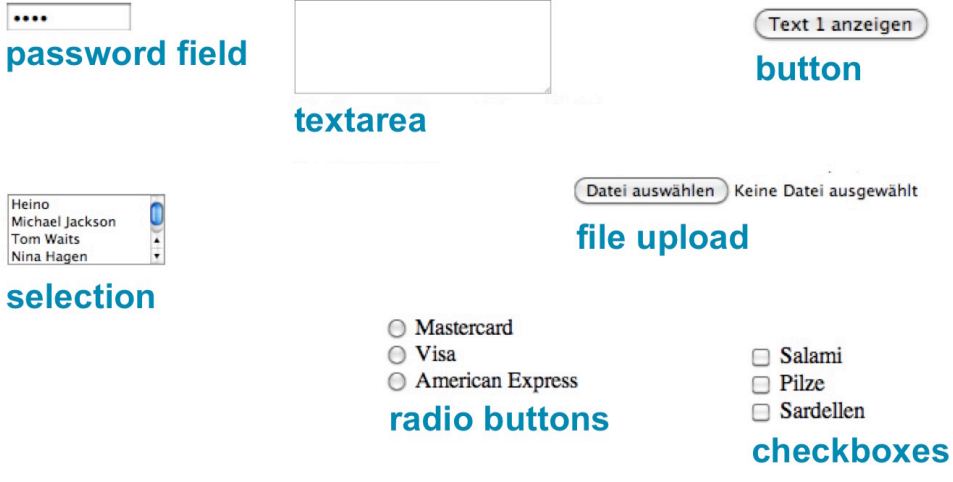
```
<form>
  <fieldset>
    <legend>General information</legend>
    <label>Text field <input type="text" value="hi"></label>
    <label>Password <input type="password" value="hi"></label>
    <label class="area">Textarea <textarea>hi</textarea></label>
  </fieldset>
  <fieldset>
    <legend>Additional information</legend>
    <label>Checkbox <input type="checkbox"></label>
    <label>Radio button <input type="radio" name="demo" checked></la
    <label>Another one <input type="radio" name="demo"></label>
  </fieldset>
  <label>Button <button>Click me</button></label>
  <label>Select menu
    <select name="cars">
      <option value="volvo">Volvo</option>
      <option value="saab">Saab</option>
      <option value="fiat">Fiat</option>
      <option value="audi">Audi</option>
    </select>
  </label>
  <input type="submit" value="Send">
</form>
```



Speaker notes

- Formularelemente können mit `fieldset` gruppiert werden
- Solche Gruppen können mit `legend` beschrieben werden

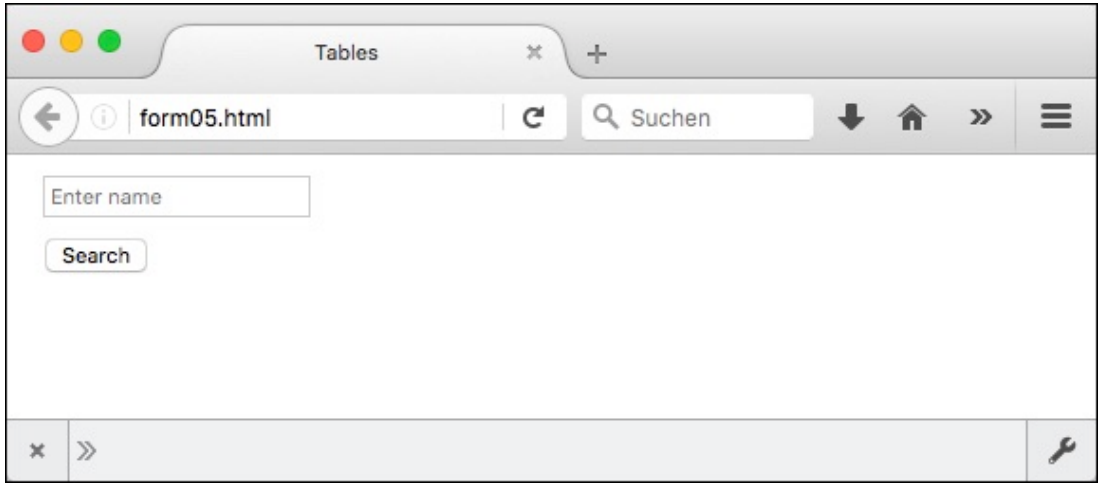
Alte Formularelemente (HTML4)



Neuere Möglichkeiten

```
<form>
  <input name="q" placeholder="Enter name">
  <input type="submit" value="Search">
</form>
```

- Placeholder-Text erscheint im leeren Eingabefeld
- Er verschwindet, sobald das Eingabefeld Text enthält



DAS type-ATTRIBUT VON input

- Spezielle Tastaturlayouts für Mobilgeräte möglich
- Werte: `email`, `url`, `range`, `date`, `search`, `color`, ...

FOKUS

- Aktives Element
- Bei Textfeldern: das welches Tastatureingaben aufnimmt
- Referenz in `document.activeElement`
- Anpassbar mit Methoden `focus` und `blur`

```
<input type="text">
<script>
  document.querySelector("input").focus()
  console.log(document.activeElement.tagName)     // → INPUT
  document.querySelector("input").blur()
  console.log(document.activeElement.tagName)     // → BODY
</script>
```

Speaker notes

Wenn beim Laden des Dokuments ein Element bereits den Fokus erhalten soll, kann dem Element ein `autofocus`-Attribut hinzugefügt werden.

Normalerweise kann der Fokus von Element zu Element mit der Tab-Taste verschoben werden. Das betrifft zunächst Formularelemente. In welcher Reihenfolge sie ausgewählt werden, lässt sich mit dem Attribut `tabindex` steuern. Mit `tabindex` können auch andere Elemente fokussierbar gemacht werden.

INAKTIVE ELEMENTE

- Attribut `disabled` (braucht keinen Wert)
- Element ist nicht verwendbar und nicht fokussierbar
- Darstellung als inaktiv

```
<button>I'm all right</button>
<button disabled>I'm out</button>
```

I'm all right I'm out

FORMULAR-EVENTS

- Formularelement geändert:
  `change`
- Eingabe in Textfeld:
  `input`
- Tastendrücke in aktivem Textfeld:
  `keydown`, `keypress`, `keyup`
- Formular absenden:
  `submit`

FILE-INPUT: DATEI WÄHLEN

- Ursprünglich: Upload einer Datei zum Server
- Mittlerweile auch per JavaScript zugänglich
- Sicherheit: Datei muss aktiv ausgewählt werden

```
<input type="file">
<script>
  let input = document.querySelector("input")
  input.addEventListener("change", () => {
    if (input.files.length > 0) {
      let file = input.files[0]
      console.log("You chose ", file.name)
      if (file.type) console.log("It has type ", file.type)
    }
  })
</script>
```

Speaker notes

Das Attribut `files` enthält ein Array-ähnliches Objekt mit Informationen zu den ausgewählten Dateien. Mit dem HTML-Attribut `multiple` kann erlaubt werden, mehrere Dateien auszuwählen.

FILE-INPUT: DATEI LESEN

```
<input type="file" multiple>
<script>
  let input = document.querySelector("input")
  input.addEventListener("change", () => {
    for (let file of Array.from(input.files)) {
      let reader = new FileReader()
      reader.addEventListener("load", () => {
        console.log("File", file.name, "starts with",
                    reader.result.slice(0, 20))
      })
      reader.readAsText(file)
    }
  })
</script>
```

Speaker notes

Das Einlesen erfolgt asynchron, damit es die Event Loop nicht blockiert. Auf dem FileReader wird ein `load`-Event registriert, das ausgelöst wird, wenn die Datei geladen ist. Sie steht nun im `result`-Attribut des FileReader zur Verfügung.

Neben dem `load`-Event kann beim Lesen auch ein `error`-Event auftreten.

Hier eine Variante, in der das Einlesen in eine Promise verpackt wurde:

```
function readFileText (file) {
  return new Promise((resolve, reject) => {
    let reader = new FileReader()
    reader.addEventListener(
      "load", () => resolve(reader.result))
    reader.addEventListener(
      "error", () => reject(reader.error))
    reader.readAsText(file)
  })
}
```

FORMULARDATEN ÜBERTRAGEN

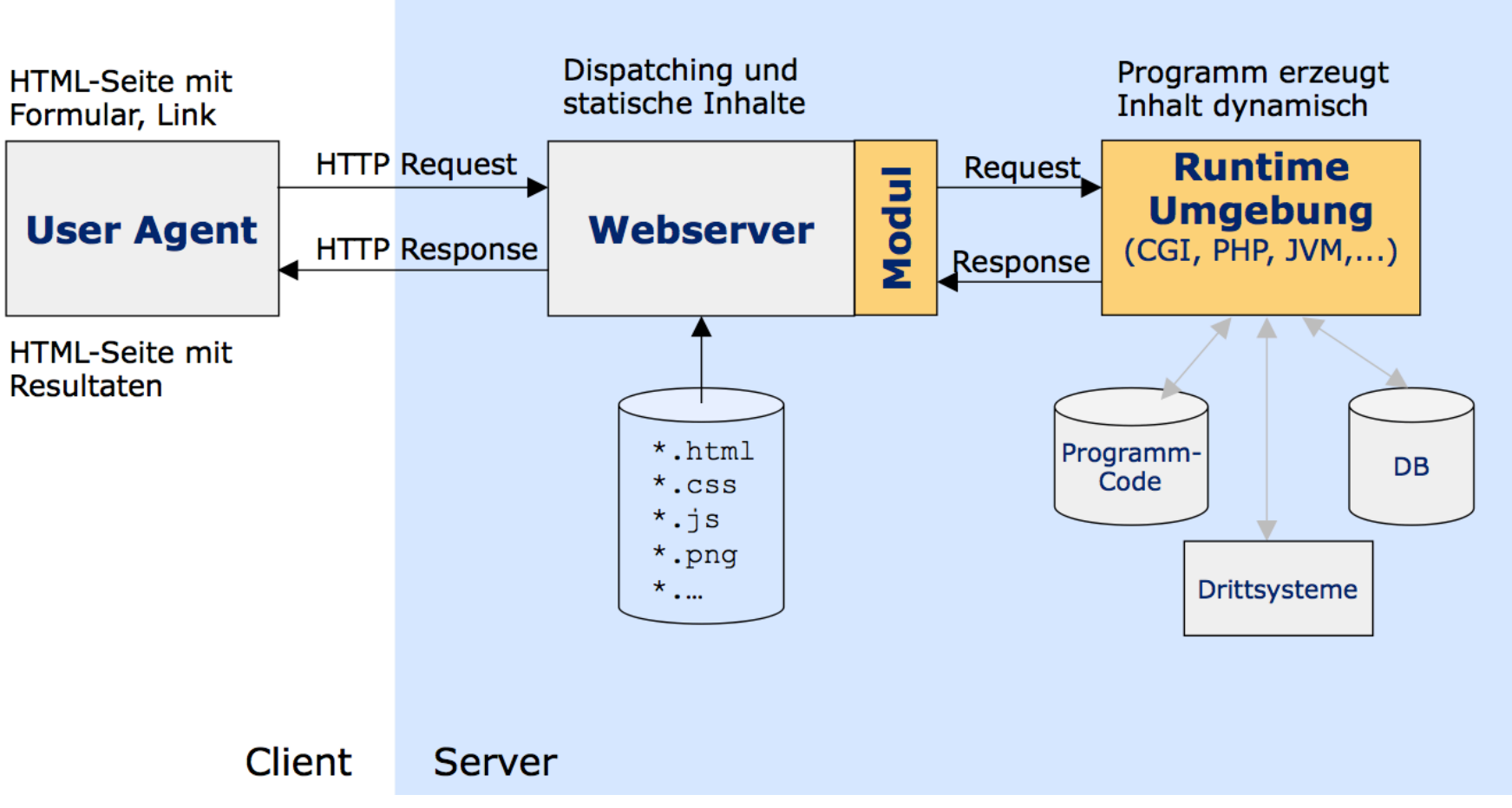
- Daten via HTTP an den Server übertragen
- Erforderlich: serverseitige Programmlogik
- Alternative: clientseitige Verarbeitung von Formulardaten
- Absenden mit Submit-Button `type="submit"`
- Löst ein `submit`-Event aus
- Im Event Handler kann das Absenden verhindert werden (`preventDefault`)

Speaker notes

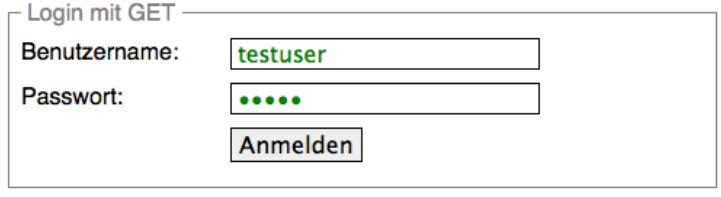
Beispiel:

```
<form action="example/submit.html">
  Value: <input type="text" name="val">
  <button type="submit">Save</button>
</form>



<script>
  let form = document.querySelector("form");
  form.addEventListener("submit", event => {
    console.log("Saving value", form.elements.val.value);
    event.preventDefault();
  });
</script>
```

SERVERSEITIGE PROGRAMMLOGIK



GET-METHODE




```
<form action="http://localhost/cgi/showenv.cgi"
      method="get">
  <fieldset>
    <legend>Login mit GET</legend>
    <label for="login_get">Benutzername:</label>
    <input type="text" id="login_get" name="login"/>
    <label for="password_get">Passwort:</label>
    <input type="password" id="password_get" name="password"/>
    <label for="submit_get"></label>
    <input type="submit" id="submit_get" name="submit" value="Anmelden" />
  </fieldset>
</form>
```

```
GET /cgi/showenv.cgi?login=testuser&password=12345%3F&submit=Anmelden HTTP/1.1
Host            localhost
User-Agent      Mozilla/5.0 (Macintosh; ...) Gecko/20100101 Firefox/31.0
Accept          text/html,application/xhtml+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
```

GET-METHODE

```
GET /cgi/showenv.cgi?login=testuser&password=12345%3F&submit=Anmelden HTTP/1.1
```

- Formulardaten an URL angehängt (Browser-Adressleiste)
- Genannt: Query String (Teil der URL)
- Vor allem für passive Anfragen (zum Beispiel eine Suche)
- Im Beispiel: `%3F` steht für `?` (URL encoding)
- Funktionen: `encodeURIComponent`, `decodeURIComponent`
- Die GET-Methode ist ungeeignet für Anmeldedaten und andere sensitiven Informationen

Speaker notes

```
console.log(encodeURIComponent("Yes?"));      // → Yes%3F
console.log(decodeURIComponent("Yes%3F"));    // → Yes?
```

POST-METHODE

- Im Formular wird `method="post"` gesetzt
- Formulardaten im Body des HTTP-Request gesendet
- Um Daten anzulegen oder zu aktualisieren
- Für sensitive Daten (Login): POST und HTTPS

```
POST /~bkrt/cgi/showenv.cgi HTTP/1.1
Host localhost
...
Content-Type application/x-www-form-urlencoded
Content-Length 44

login=testuser&password=1234%3F&submit=Anmelden
```

GET: EXPRESS

```
// GET /shoes?order=desc&shoe[color]=blue&shoe[type]=converse
console.dir(req.query.order)
// => 'desc'

console.dir(req.query.shoe.color)
// => 'blue'

// route: /user/:name
// GET /user/tj
console.dir(req.params.name)
// => 'tj'
```

Speaker notes

Zum Vergleich: PHP

```php
<?php
  // So nicht: Login-Daten per GET
  $user = $_GET['login'];
  $pw =   $_GET['password'];
  ...
?>
```

Zum Vergleich: JSP

```jsp
<%
  // GET und POST
  String user = request.getParameter("login");
  String pw   = request.getParameter("password");
%>
```

POST: EXPRESS

```
var express = require('express')
var app = express()

// for parsing application/json
app.use(express.json())
// for parsing application/x-www-form-urlencoded
app.use(express.urlencoded({ extended: true }))

app.post('/profile', function (req, res, next) {
  console.log(req.body)
  res.json(req.body)
})
```

Speaker notes

Zum Vergleich: PHP

```php
<?php
    // Login-Daten per POST
    $user = $_POST['login'];
    $pw =   $_POST['password'];
    ...
?>
```

ÜBERSICHT

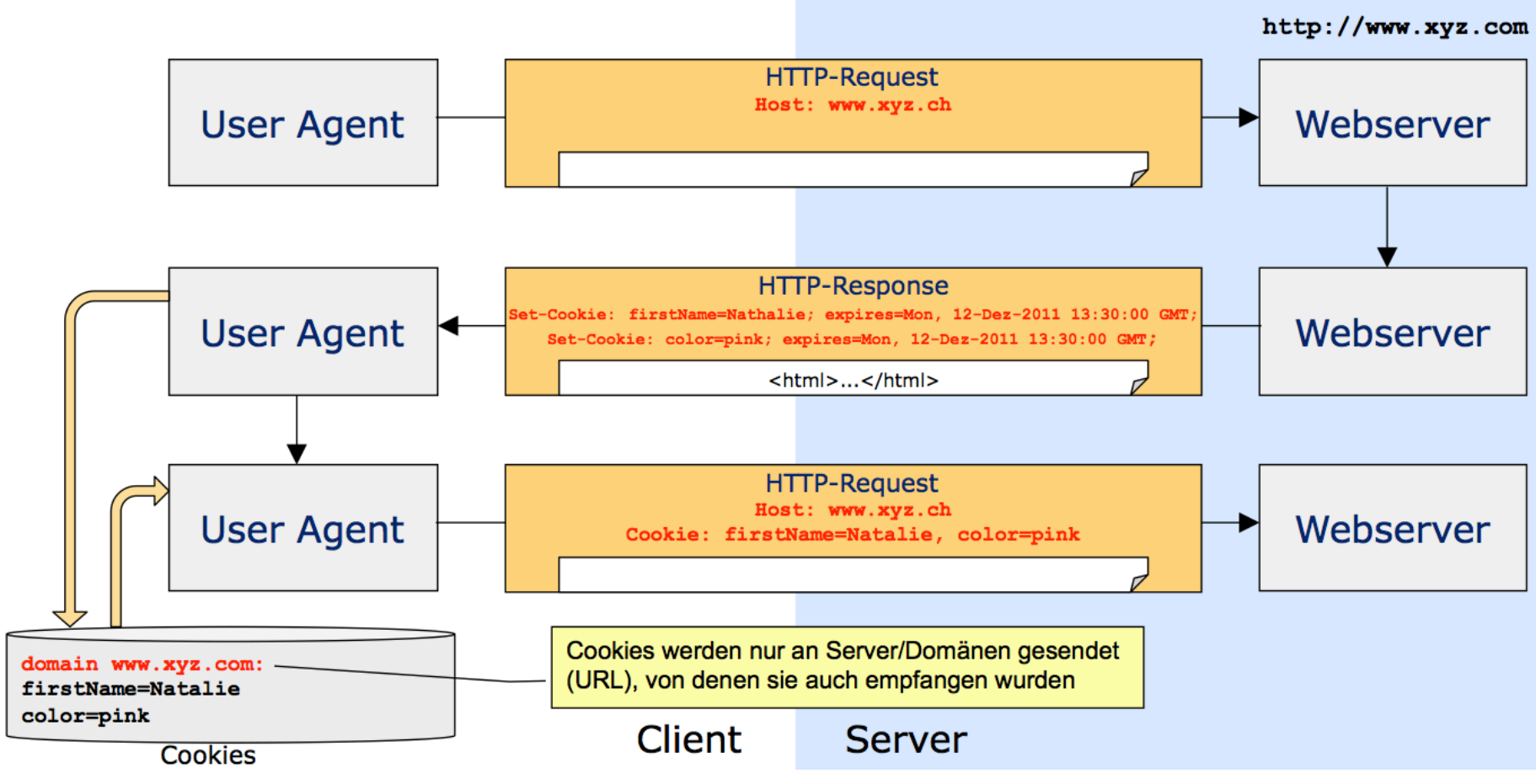
- Formulare
- Cookies, Sessions
- Ajax und XMLHttpRequest
- Fetch API

COOKIES

- HTTP als zustandsloses Protokoll konzipiert
- Cookies: Speichern von Informationen auf dem Client
- RFC 2965: HTTP State Management Mechanism
- Response: `Set-Cookie`-Header, Request: `Cookie`-Header
- Zugriff mit JavaScript möglich (ausser `HttpOnly` ist gesetzt)

```
allCookies = document.cookie
```

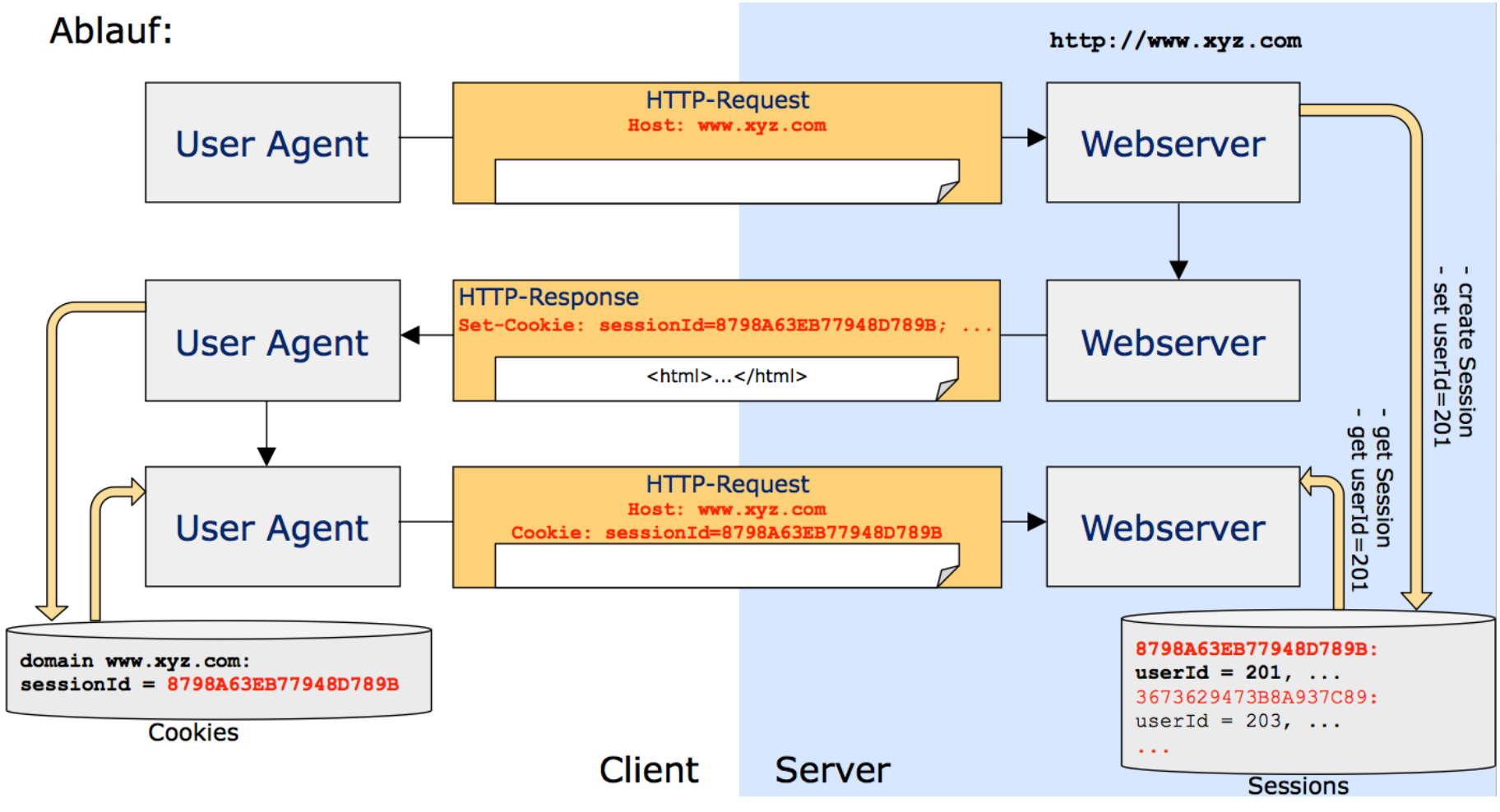
COOKIES


http://www.xyz.com
User Agent
HTTP-Request
Host: www.xyz.ch
Webserver
User Agent
HTTP-Response
Set-Cookie: firstName=Nathalie; expires=Mon, 12-Dez-2011 13:30:00 GMT;
Set-Cookie: color=pink; expires=Mon, 12-Dez-2011 13:30:00 GMT;
<html>...</html>
Webserver
User Agent
HTTP-Request
Host: www.xyz.ch
Cookie: firstName=Natalie, color=pink
Webserver
domain www.xyz.com:
firstName=Natalie
color=pink
Cookies
Cookies werden nur an Server/Domänen gesendet (URL), von denen sie auch empfangen wurden
Client
Server

SESSIONS

- Cookies auf dem Client leicht manipulierbar
- Session: Client-spezifische Daten auf dem Server speichern
- Identifikation des Clients über Session-ID (Cookie o.a.)
- Gefahr: Session-ID gerät in falsche Hände (Session-Hijacking)

SESSIONS


Ablauf:
http://www.xyz.com
User Agent
HTTP-Request
Host: www.xyz.com
Webserver
- create Session
- set userId=201
User Agent
HTTP-Response
Set-Cookie: sessionId=8798A63EB77948D789B; ...
<html>...</html>
Webserver
- get Session
- get userId=201
User Agent
HTTP-Request
Host: www.xyz.com
Cookie: sessionId=8798A63EB77948D789B
Webserver
domain www.xyz.com:
sessionId = 8798A63EB77948D789B
Cookies
8798A63EB77948D789B:
userId = 201, ...
3673629473B8A937C89:
userId = 203, ...
...
Sessions
Client
Server

SESSIONS: EXPRESS

```
$ npm install express-session

const express = require('express')
const cookieParser = require('cookie-parser')
const session = require('express-session')
const app = express();
app.use(cookieParser())
app.use(session({secret: "Shh, its a secret!"}))

app.get('/', function(req, res){
   if(req.session.page_views){
      req.session.page_views++
      res.send("You visited this page " + req.session.page_views + " times")
   } else {
      req.session.page_views = 1
      res.send("Welcome to this page for the first time!")
   }
})
app.listen(3000)
```

Speaker notes

Beispiel aus:
https://www.tutorialspoint.com/expressjs/expressjs_sessions.htm

Zum Vergleich: PHP

```php
<?php
  session_start();

  // Session-Variable setzen
  $_SESSION['userid'] = '101';
?>
```

```php
<?php
  session_start();

  // Session-Variable auslesen
  $uid = $_SESSION['userid'];
?>
```

Zum Vergleich: JSP

```
<%
  // int automatisch nach Integer konvertiert (auto-boxing)
  session.setAttribute("userid", 101);
%>

<%
  // Typkonvertierung, da getAttribute() den Typ Object liefert
  Integer userid = (Integer) session.getAttribute("userid");
%>
```

ÜBERSICHT

- Formulare
- Cookies, Sessions
- Ajax und XMLHttpRequest
- Fetch API

TRADITIONELLES WEB-MODELL

- Seite für Seite via HTTP in Browser laden
- Für Netz von Dokumenten gut geeignet
- Für interaktive Applikationen weniger

- Idee: Informationen asynchron im Hintergrund laden
- Technische Grundlage: `XMLHttpRequest`-Objekt

Speaker notes

Auch ohne das `XMLHttpRequest`-Objekt konnte man Informationen im Hintergrund nachladen, das war aber sehr umständlich und teilweise mit unschönen Nebeneffekten verbunden. Zum Beispiel konnte man die Kommunikation im Hintergrund über ein in die Seite eingebettetes *IFrame* durchführen. Auch wenn man diesem IFrame eine Breite und Höhe von 0 gab, war es trotzdem auf der Seite vorhanden und konnte via JavaScript mit der umgebenden Webseite kommunizieren.

BEZEICHNUNG: Ajax

Jesse James Garrett, 2005:
Ajax: A New Approach to Web Applications

Ajax isn’t a technology. It’s really several technologies, each flourishing in its own right, coming together in powerful new ways. It incorporates:

- standards-based presentation using XHTML and CSS;
- dynamic display and interaction using the DOM;
- data interchange and manipulation using XML and XSLT;
- asynchronous data retrieval using XMLHttpRequest;
- and JavaScript binding everything together.

Speaker notes

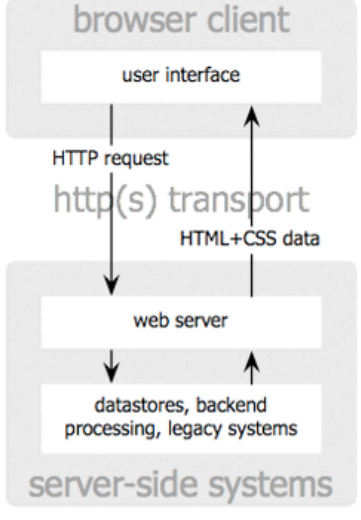
Das Original-Dokument ist nicht mehr online. Man kann es aber via *archive.org* noch zugreifen:

https://web.archive.org/web/20160110132554/http://adaptivepath.org/ideas/ajax-new-approach-web-applications/

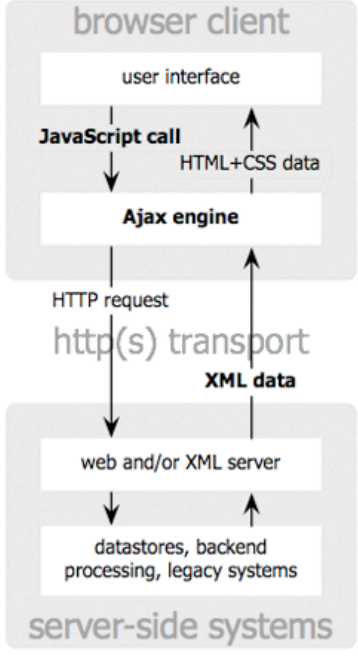
Statt XML und XSLT wird heute meist JSON für den Datenaustausch verwendet.

https://www.w3schools.com/js/js_json_xml.asp

AJAX



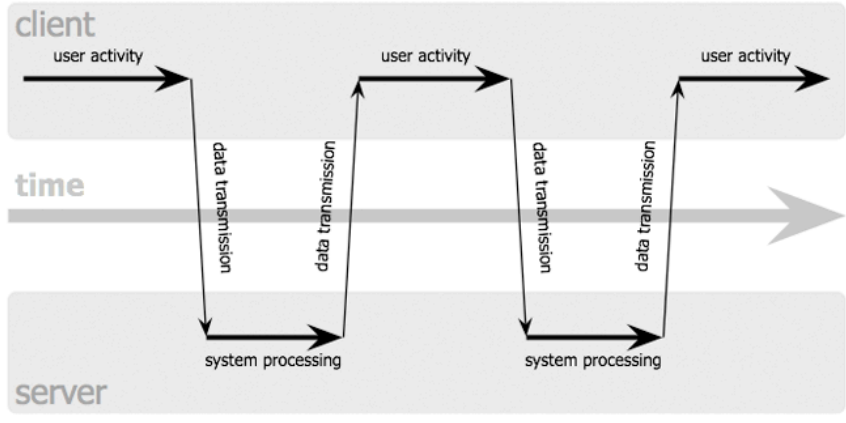
classic
web application model



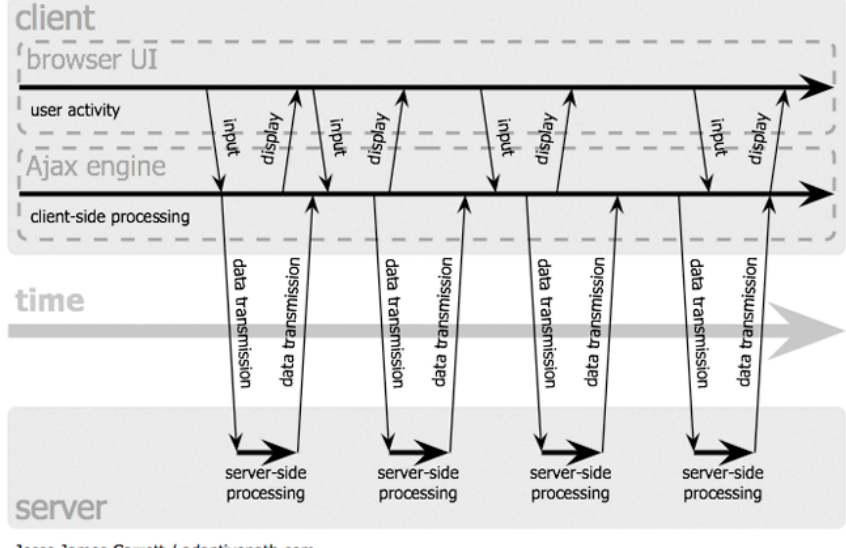
Ajax
web application model

Jesse James Garrett / adaptivepath.com

classic web application model (synchronous)



Ajax web application model (asynchronous)



Jesse James Garrett / adaptivepath.com

XMLHTTPREQUEST: BEISPIEL

```
const xhr = new XMLHttpRequest()
xhr.onreadystatechange = () => {
  if (xhr.readyState === 4) {
    xhr.status === 200 ? console.log(xhr.responseText)
                       : console.error('error')
  }
}
xhr.open('GET', 'https://yoursite.com')
xhr.send()
```

- Low-level, umständlich einzusetzen
- Besser Abstraktionen wie Fetch oder Axios verwenden
- XMLHttpRequest Spec: https://xhr.spec.whatwg.org

Speaker notes

Das Beispiel stammt aus: "The Node.js Handbook" (Flavio Copes).

`XMLHttpRequest` wurde ursprünglich von Microsoft in *Outlook Web Access* eingeführt, war aber lange kein Standard. Glücklicherweise müssen wir uns heute nicht mehr mit der Arbeitsweise von `XMLHttpRequest` herumschlagen, sondern können die komfortable Fetch-API (oder die entsprechenden Funktionen von Node.js oder jQuery) verwenden.

ÜBERSICHT

- Formulare
- Cookies, Sessions
- Ajax und XMLHttpRequest
- Fetch API

FETCH API

- Interface for fetching resources
- Gibt Promise zurück
- Nach Server-Antwort erfüllt mit Response-Objekt

```
cons url = 'https://...'

fetch(url).then(response => {
  console.log(response.status)                    // → 200
  console.log(response.headers.get("Content-Type"))    // → text/plain
})
```

Speaker notes

Da das Fetch API noch nicht so alt ist, verwendet es Promises. Das ist bei den meisten Browser APIs nicht der Fall.

Auch wenn der Server mit einem Fehlercode antwortet, ist das Promise-Objekt *resolved*. Im Fall eines Netzwerkfehlers oder der Server aus anderen Gründen nicht erreichbar ist, kann die Promise aber auch *rejected* werden.

Erstes Argument von `fetch` ist die URL. Statt einer kompletten URL kann auch ein relativer oder absoluter Pfad angegeben werden, der sich auf die aktuelle Seite bezieht.

RESPONSE-OBJEKT

- `headers`: Zugriff auf HTTP-Header-Daten
  Methoden `get`, `keys`, `forEach`, ...
- `status`: Status-Code
- `json()`: liefert Promise mit Resultat der JSON-Verarbeitung
- `text()`: liefert Promise mit Inhalt der Server-Antwort

FETCH API

```
fetch("example/data.txt")
  .then(resp => resp.text())
  .then(text => console.log(text))
// → This is the content of data.txt
```

- Promise des `fetch`-Aufrufs ist bereits nach Verarbeiten des Headers erfüllt
- Laden (und Verarbeiten) des Inhalts dauert aber länger
- Daher retournieren `json()` und `text()` selbst eine Promise
- Promise von `json()` bei Parse-Error rejected

ANDERE HTTP-METHODEN

- Normalerweise macht `fetch` eine `GET`-Anfrage
- Als zweites Argument Objekt mit weiteren Einstellungen möglich
  - `method`: HTTP-Methode
  - `headers`: HTTP Headers ergänzen
  - `body`: Inhalt mitsenden

```
fetch("example/data.txt", {method: "DELETE"}).then(resp => {
  console.log(resp.status)
  // → 405
})
```

Speaker notes

Weiteres Beispiel:

```
fetch("example/data.txt", {headers: {Range: "bytes=8-19"}})
  .then(resp => resp.text())
  .then(console.log);
// → the content
```

Mit dem Range-Header kann ein Teil (oder mehrere Teile) eines Dokuments angefordert werden.

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

ZUM VERGLEICH: JQUERY

```
$.ajax({
  url: "/api/getWeather",
  data: {
    zipcode: 97201
  },
  success: function (result) {
    $("#weather-temp").html("<strong>" + result + "</strong> degrees")
  }
})
```

Speaker notes

Die gezeigte Funktion `$.ajax` ist das Low-Level Interface von jQuery für Ajax mit den meisten Möglichkeiten. Für bestimmte Zwecke gibt es eine Reihe von speziellen Methoden, die einfacher zu verwenden sind:

- `$.get`: GET-Request ausführen
- `$.getJSON`: GET-Request ausführen und Ergebnis als JSON parsen
- `$.post`: POST-Request ausführen
- `.load`: (an Elementknoten ausgeführt) HTML-Code laden und beim Element einfügen

http://jqapi.com

ALTERNATIVE ZU FETCH: AXIOS

- HTTP-Client für Browser und Node.js
- Ebenfalls basierend auf Promises

```
async function getUser() {
  try {
    const response = await axios.get('/user?ID=12345');
    console.log(response);
  } catch (error) {
    console.error(error);
  }
}
```

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

SICHERHEIT

- Browser blockieren normalerweise Zugriffe von Scripts auf andere Domains
- Genannt: same-origin Policy
- Grund: Angriffsmöglichkeiten reduzieren
- Manchmal ist der Zugriff auf andere Server erwünscht
- Cross-Origin Resource Sharing (CORS)
- Server kann Zugriff erlauben mit diesem Header:

```
Access-Control-Allow-Origin: *
```

Speaker notes

Warum werden durch die *same-origin Policy* Angriffe reduziert? Angenommen, Sie kommen versehentlich auf eine Website `themafia.org`. Dann möchten Sie normalerweise nicht, dass Scripts, die von dieser Website geladen werden, Zugriff auf `mybank.com` haben. Dies könnte besonders dann unschöne Folgen haben, wenn Sie gerade bei `mybank.com` eingeloggt sind.

Zum Beispiel in Express:

```
// CORS header `Access-Control-Allow-Origin` set to accept all
app.get('/allow-cors', function(request, response) {
  response.set('Access-Control-Allow-Origin', '*');
  response.sendFile(__dirname + '/message.json');
});
```

https://en.wikipedia.org/wiki/Cross-origin_resource_sharing

In der Vergangenheit wurden auch andere "Tricks" verwendet, um solche Cross-Origin-Anfragen zu ermöglichen. Eine davon ist JSON-P (JSON with Padding), die zum Laden von Daten ein `script`-Tag anlegt und den Server anfragt, die gelieferten Daten in einen Funktionsaufruf zu verpacken. Das sollte heute nicht mehr verwendet werden.

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

HTTPS

- Verschlüsselte Übertragung mittels SSL/TLS
- Unerlässlich für alle Arten sensitiver Daten
- Der Standard-Port ist 443
- Server stellt Zertifikat zur Verfügung
- Browser muss dieses überprüfen können

https://de.wikipedia.org/wiki/Hypertext_Transfer_Protocol_Secure

QUELLEN

- Marijn Haverbeke: Eloquent JavaScript, 3rd Edition
  https://eloquentjavascript.net/
- Ältere Slides aus WEB2 und WEB3

LESESTOFF

Geeignet zur Ergänzung und Vertiefung

- Kapitel 18 von:
  Marijn Haverbeke: Eloquent JavaScript, 3rd Edition
  https://eloquentjavascript.net/
- HTML Forms (w3schools)
  http://www.w3schools.com/html/html_forms.asp
- Building Forms (Shay Howe, Learn to Code HTML & CSS)
  http://learn.shayhowe.com/html-css/building-forms/
- A Form Of Madness (Mark Pilgrim, Dive Into HTML5)
  http://diveinto.html5doctor.com/forms.html