

JavaScript im Einsatz auf der Client-Seite

AJAX, DOM, FETCH

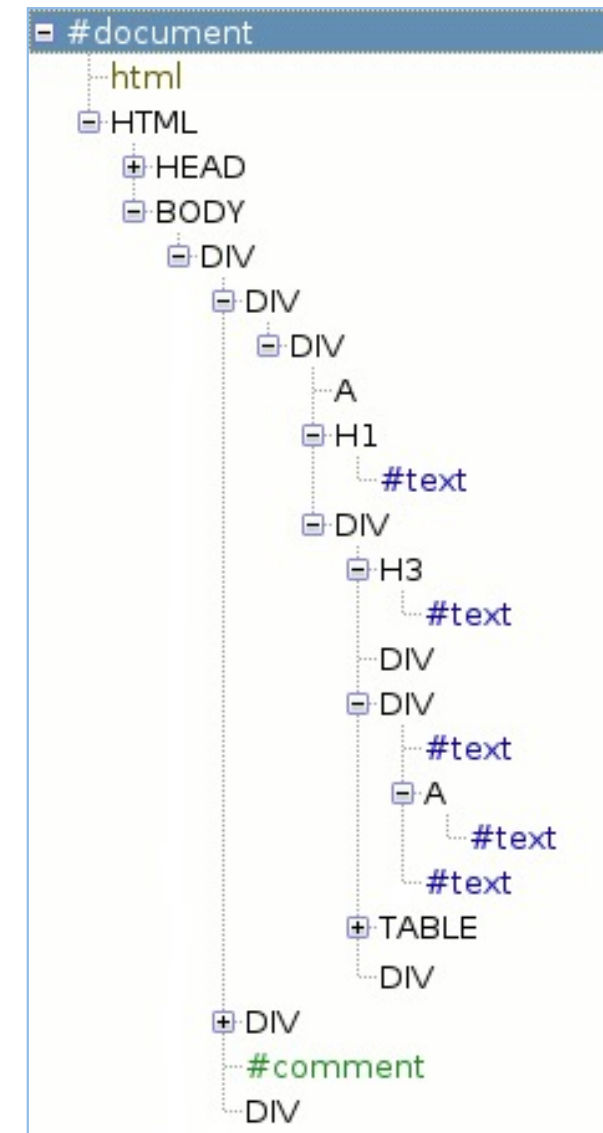
Document Object Model

Bietet Zugriff auf alle Tags und Attribute der Webseite

- Baumstruktur
- Erlaubt Manipulation des vorliegenden HTML-Dokuments

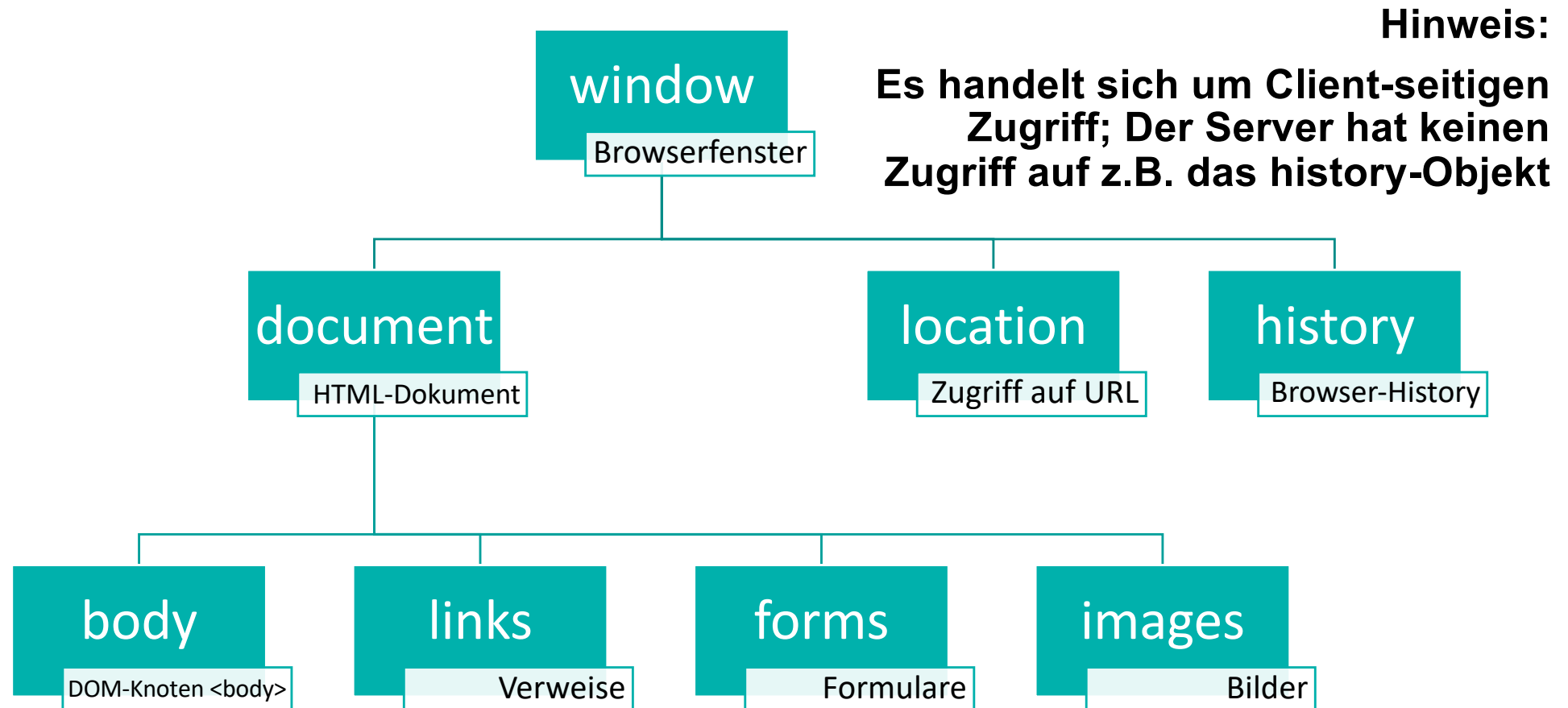
„Dunkle Vergangenheit“

- JavaScript war zwar standardisiert, nicht aber der Zugriff auf das HTML-Dokument
 - > Jeder Browser regelte den Zugriff anders
- Mittlerweile standardisiert
 - > Teilweise immer noch Unterschiede



Document Object Model

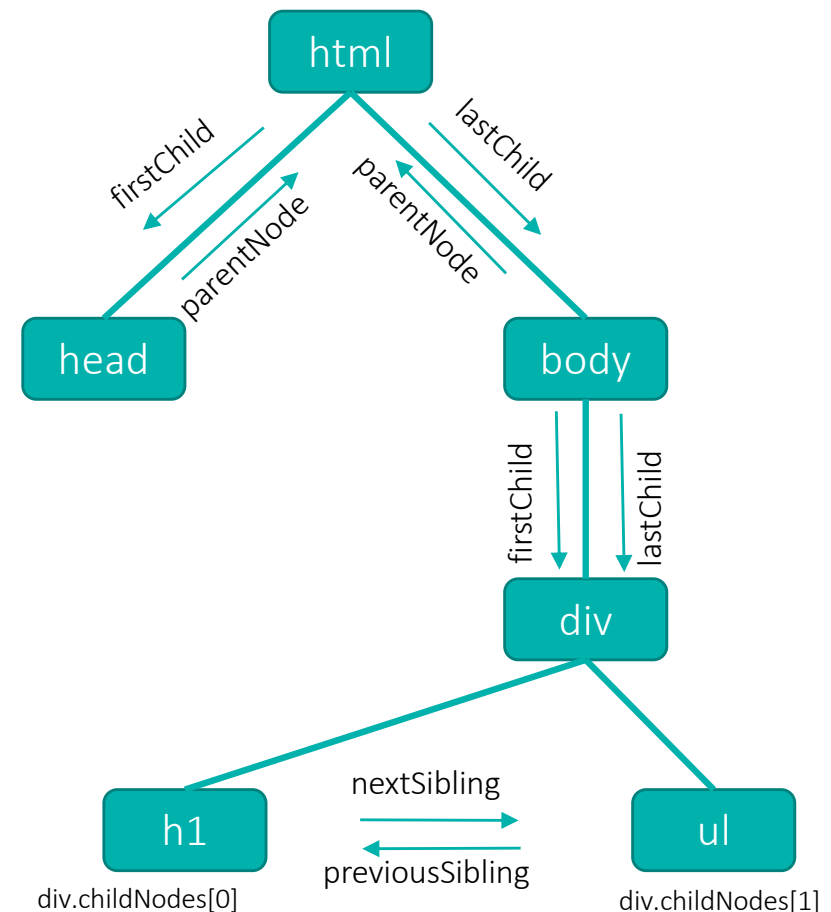
Ausschnitt der relevanten Objekte:



Document Object Model

Zugriff auf DOM-Knoten

- über ihre id (einfachste Möglichkeit) oder per Klasse/Tag/Name
- über die numerische Ordnung in der Hierarchie durch Beschreiten des entsprechenden Feldes, das die Objekte beinhaltet
- über die Position im DOM-Baum und dem entsprechenden Navigieren (parentNode, previousSibling, nextSibling, firstChild, lastChild, childNodes)



Document Object Model

Beispiel

```
<div id="meindiv">
```

```
Dies ist ein einfacher Text
```

```
</div>
```

- Wir referenzieren **ein** Element mit Auszeichnungselement div mit der id “meindiv”:

```
document.getElementById('meindiv')
```

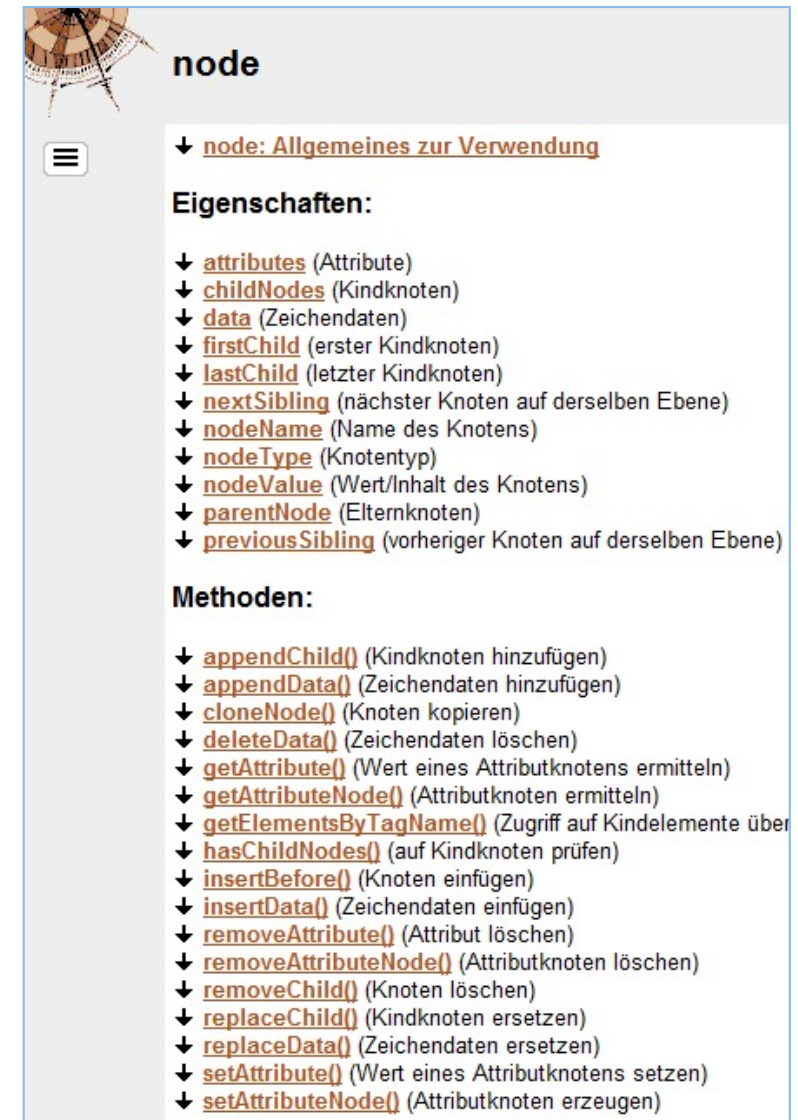
- Es beinhaltet ein Textelement, welches im obigen Teil auch über `div.childNodes[0]` referenziert werden könnte (childNodes ist ein Feld in dem alle Kindelemente gelistet werden)
- Der Text “Dies ist ein einfacher Text” ist somit kein Wert des div-Elements, vielmehr ist es der Wert des ersten und einzigen Kindelementes von div
- Zugriff: `str = document.getElementById('meindiv').innerHTML`
`str -> Dies ist ein einfacher Text`
- Ändern: `document.getElementById('meindiv').innerHTML = 'neu'`

Document Object Model

Zugriff auf Elemente

```
document.getElementById(id)
```

- Liefert den DOM-Knoten mit der ID id zurück (oder null)
- Viele andere Funktionen um Zugriff auf DOM-Knoten zu erhalten und um diese zu manipulieren
- Einige davon arbeiten auch für mehrere Hits



The screenshot shows the MDN web site entry for the 'node' object. It includes a title 'node', a sub-header 'node: Allgemeines zur Verwendung', a list of properties (attributes, childNodes, data, firstChild, lastChild, nextSibling, nodeName, nodeType, nodeValue, parentNode, previousSibling) with their descriptions, and a list of methods (appendChild, appendData, cloneNode, deleteData, getAttribute, getAttributeNode, getElementsByTagName, hasChildNodes, insertBefore, insertData, removeAttribute, removeAttributeNode, removeChild, replaceChild, replaceData, setAttribute, setAttributeNode) with their descriptions.

node

↓ node: Allgemeines zur Verwendung

Eigenschaften:

- ↓ attributes (Attribute)
- ↓ childNodes (Kindknoten)
- ↓ data (Zeichendaten)
- ↓ firstChild (erster Kindknoten)
- ↓ lastChild (letzter Kindknoten)
- ↓ nextSibling (nächster Knoten auf derselben Ebene)
- ↓ nodeName (Name des Knotens)
- ↓ nodeType (Knotentyp)
- ↓ nodeValue (Wert/Inhalt des Knotens)
- ↓ parentNode (Elternknoten)
- ↓ previousSibling (vorheriger Knoten auf derselben Ebene)

Methoden:

- ↓ appendChild() (Kindknoten hinzufügen)
- ↓ appendData() (Zeichendaten hinzufügen)
- ↓ cloneNode() (Knoten kopieren)
- ↓ deleteData() (Zeichendaten löschen)
- ↓ getAttribute() (Wert eines Attributknotens ermitteln)
- ↓ getAttributeNode() (Attributknoten ermitteln)
- ↓ getElementsByTagName() (Zugriff auf Kindelemente über TagName)
- ↓ hasChildNodes() (auf Kindknoten prüfen)
- ↓ insertBefore() (Knoten einfügen)
- ↓ insertData() (Zeichendaten einfügen)
- ↓ removeAttribute() (Attribut löschen)
- ↓ removeAttributeNode() (Attributknoten löschen)
- ↓ removeChild() (Knoten löschen)
- ↓ replaceChild() (Kindknoten ersetzen)
- ↓ replaceData() (Zeichendaten ersetzen)
- ↓ setAttribute() (Wert eines Attributknotens setzen)
- ↓ setAttributeNode() (Attributknoten erzeugen)

Quelle: <http://de.selfhtml.org/javascript/objekte/node.htm>

Document Object Model

Eine alte native Form der Manipulation

Beispiel

```
window.onload = function() {  
    document.getElementById('btn').onclick = function() {  
        // Neues h1-Element erstellen mit Inhalt "Klick"  
        var h1 = document.createElement('h1');  
        var text = document.createTextNode('Klick!');  
        h1.appendChild(text);  
  
        // In den body hinten einfügen  
        document.querySelector('body').appendChild(h1);  
    }  
};
```

Hinweis: document/element.querySelector ist die perfekte (leider langsame) Interaktion mit CSS

- > Das Argument ist ein gültiger CSS-Selektor und gibt die Referenz auf das erste zutreffende DOM-Element zurück
- > querySelectorAll gibt eine NodeList für alle zutreffenden Elemente zurück

Document Object Model

Wann kann die DOM-Manipulation starten?

Hinweis: hier bietet es sich an mittels des '**DOMContentLoaded**'-Ereignisses zu arbeiten

- > Das eigentliche Ziel ist das Document-Objekt
- > Da das Window-Objekt im Browser aber immer verfügbar ist bietet sich dieser Umweg an. Wir werden noch sehen, dass wir bei Fetch das Document-Object noch aktiv erfragen müssen

```
window.addEventListener('DOMContentLoaded', (event) => {  
    console.log('DOM fully loaded and parsed');  
});
```


Document Object Model

Browser-Engines

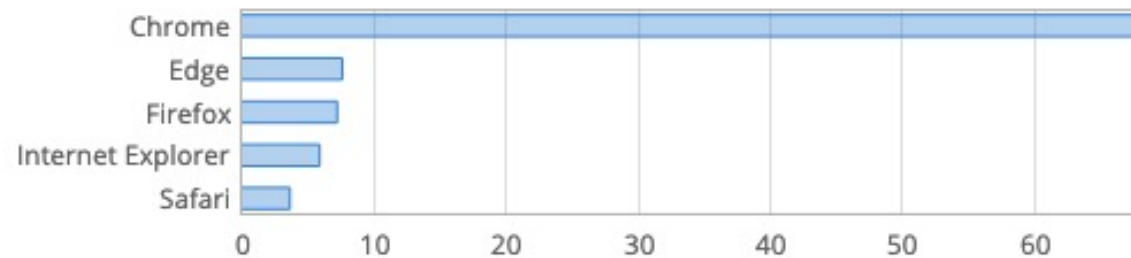
- WebKit   
 - Blink (Fork von WebKit)   
- Gecko  
- Jede Engine verhält sich ein wenig anders
 - > Vor allem veraltete Browser!

DOM-Manipulation ist kompliziert!

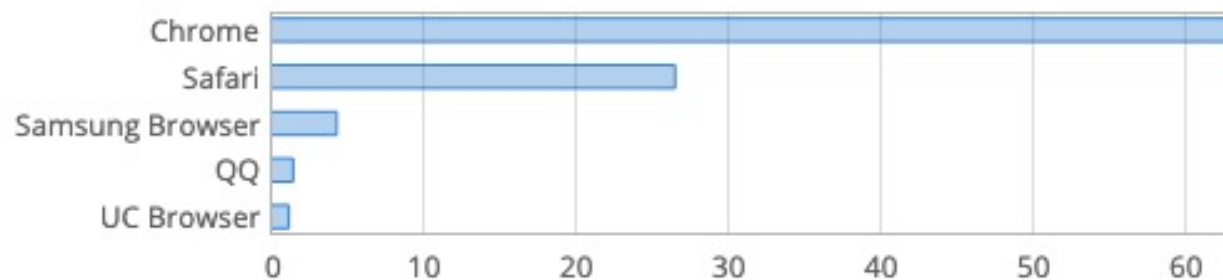
- Daher sollten Alternativen genutzt werden: (rückläufig jQuery) Fetch!

Marktanteile Browser (weltweit)

Desktop Market Share



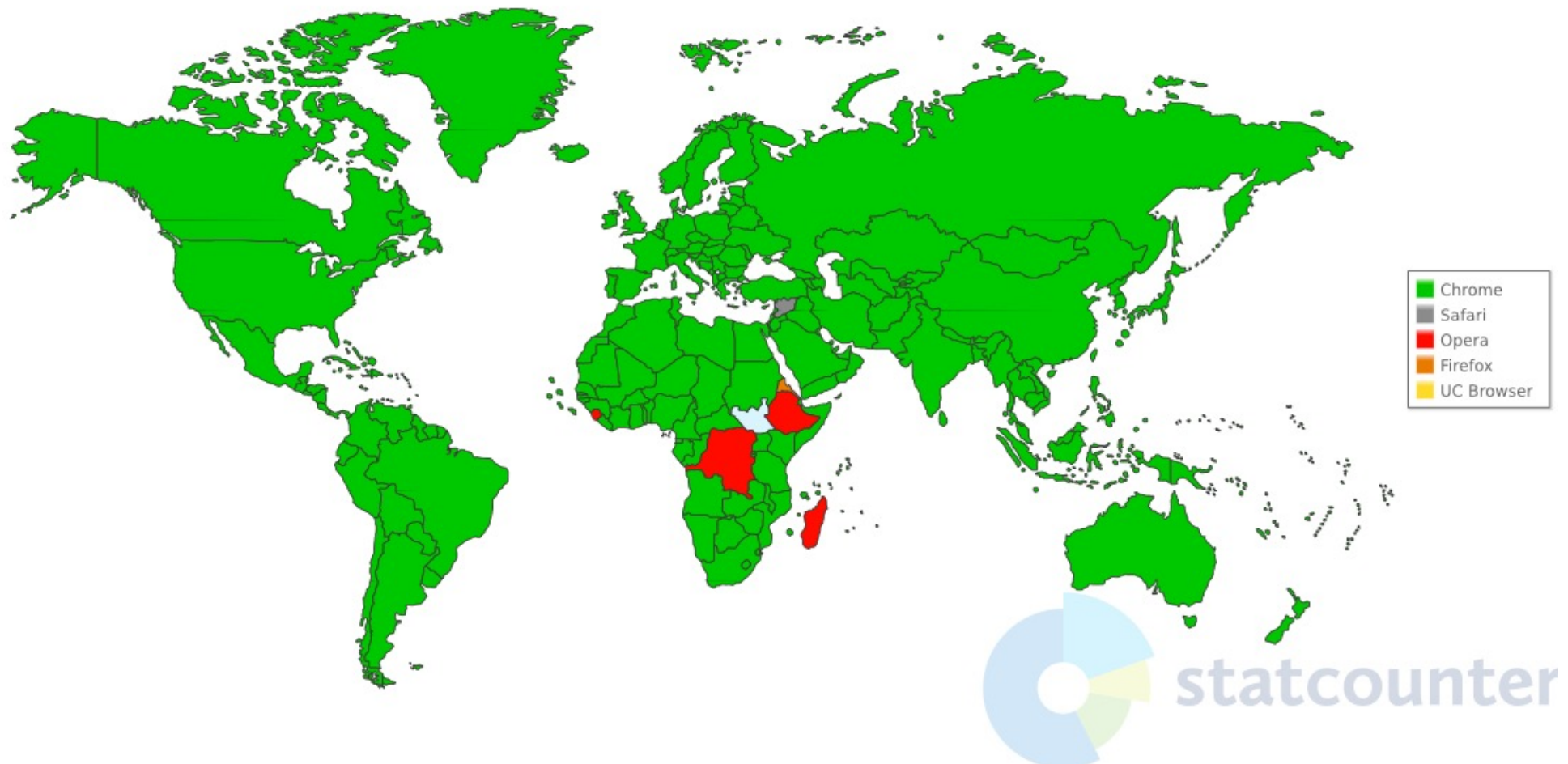
Mobile Market Share



Stand März 2020, Quelle: <https://netmarketshare.com/>

Marktanteile Browser (weltweit)

StatCounter Global Stats
Browser Market Share Worldwide, Sept 2019



Stand September 2019, Quelle: https://en.wikipedia.org/wiki/Usage_share_of_web_browsers

Woher bekomme ich meine Daten?

AJAX (Asynchronous JavaScript and XML)

Ablauf

- Benutzeraktion erzeugt JavaScript-Aufruf
- JavaScript erzeugt Daten
- Daten werden asynchron (im Hintergrund) versendet
- Mit dem Eintreffen bedingen die Daten eine Manipulation des DOMs

Typische Anwendungsgebiete

- Vorschläge für Suchbegriffe
- Webbasierte Anwendungen (Maps, Textverarbeitung, ...)
- Cloud-Anwendungen (SaaS)

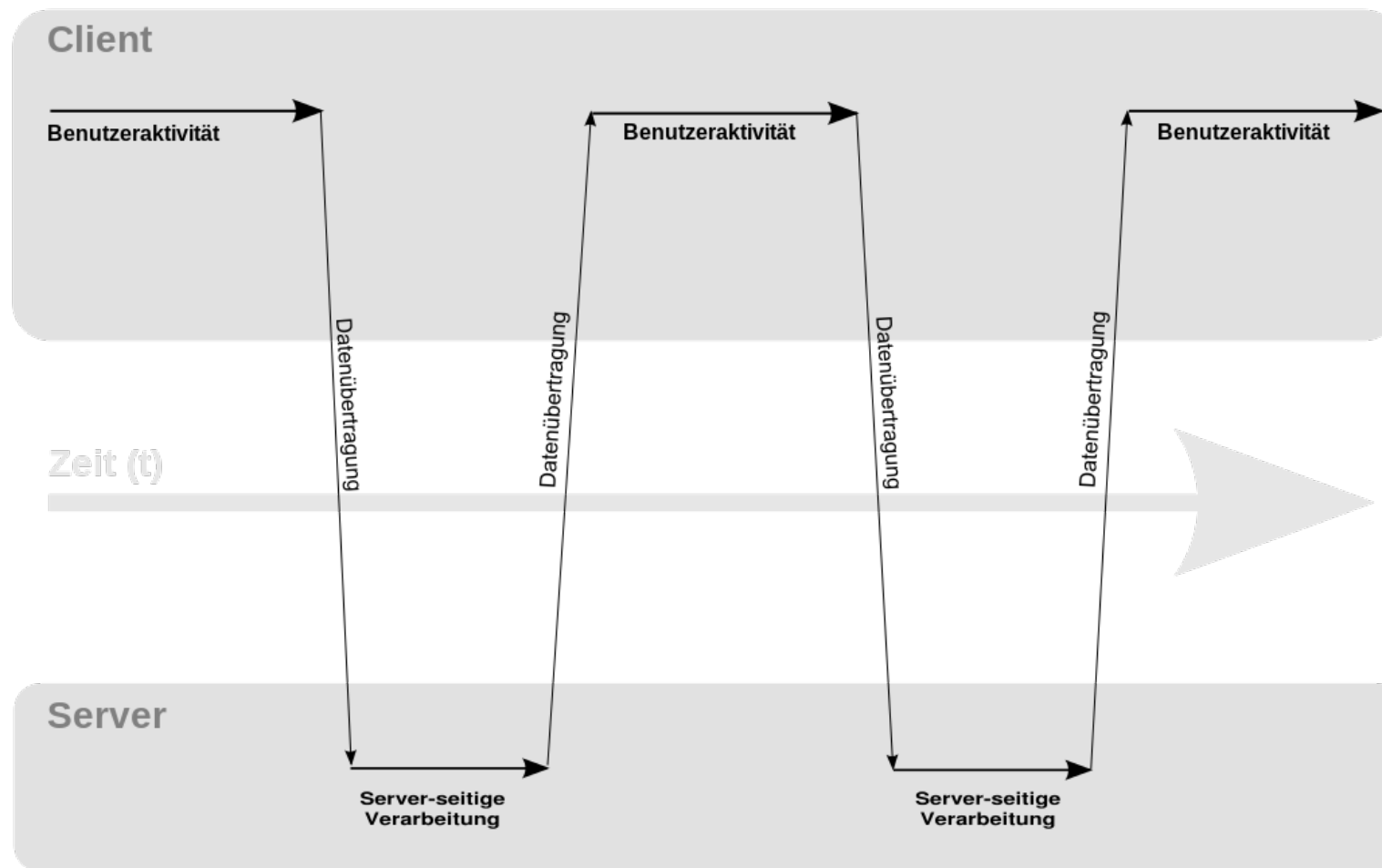
Wir sind bei AJAX!

- Faktisch ist XMLHttpRequest das ursprüngliche Herzstück von AJAX (kein direkter Zugriff auf klassische Sockets und nicht nur XML)

AJAX

Klassisches Abarbeitungsmodell

Klassisches Modell einer Web-Anwendung (synchrone Datenübertragung)

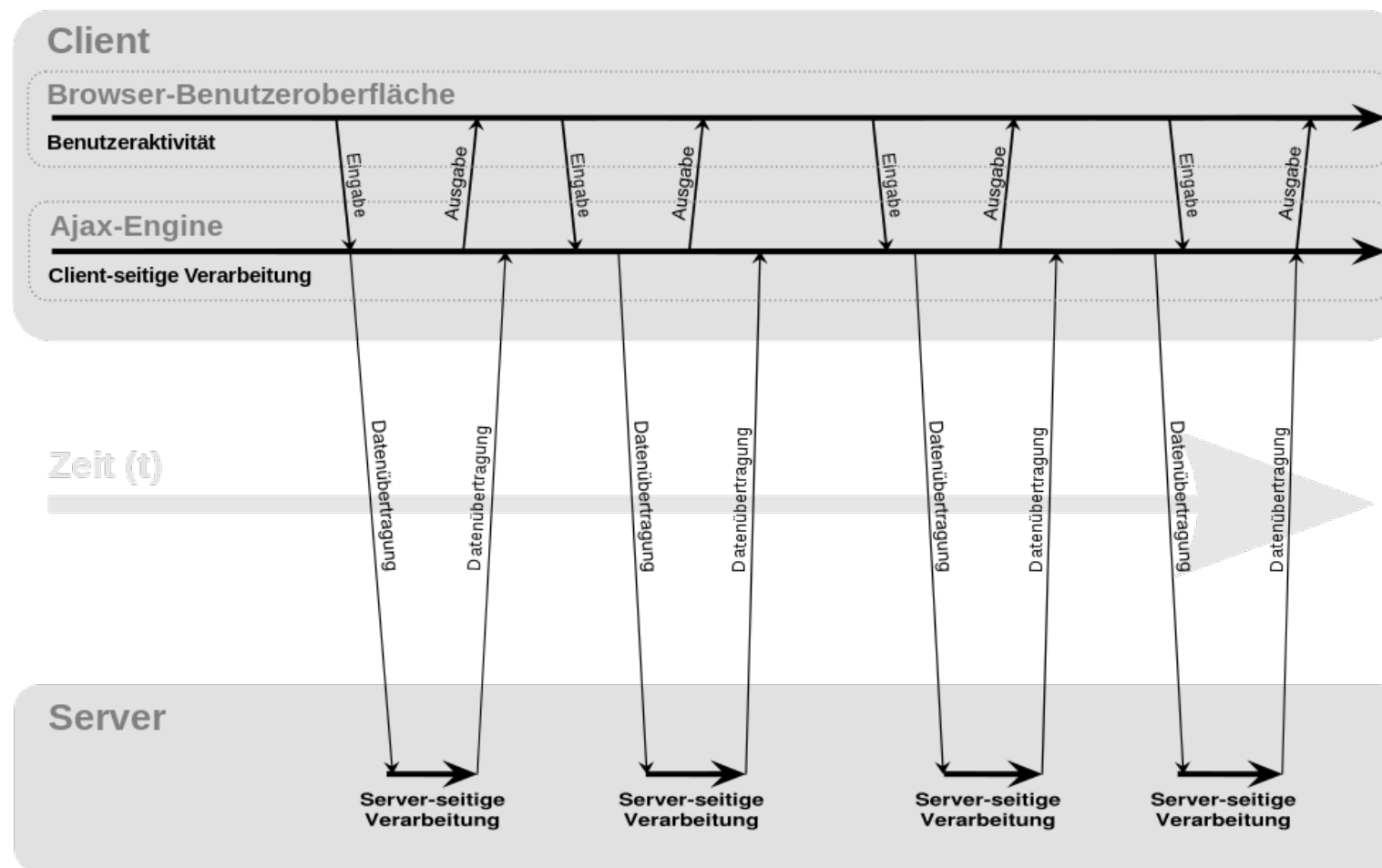


Quelle: Jesse James Garrett, Ajax: A New Approach to Web Applications, adaptive path publications, February 2005

AJAX

AJAX Abarbeitungsmodell

Ajax Modell einer Web-Anwendung (asynchrone Datenübertragung)



Quelle: Jesse James Garrett, Ajax: A New Approach to Web Applications, adaptive path publications, February 2005

War die interne Basis einer jeglichen AJAX-Interaktion

Method	Description
<code>new XMLHttpRequest()</code>	Creates a new XMLHttpRequest object
<code>abort()</code>	Cancels the current request
<code>getAllResponseHeaders()</code>	Returns header information
<code>getResponseHeader()</code>	Returns specific header information
<code>open(method,url,async,user,psw)</code>	Specifies the request <i>method</i> : the request type GET or POST <i>url</i> : the file location <i>async</i> : true (asynchronous) or false (synchronous) <i>user</i> : optional user name <i>psw</i> : optional password
<code>send()</code>	Sends the request to the server Used for GET requests
<code>send(string)</code>	Sends the request to the server. Used for POST requests
<code>setRequestHeader()</code>	Adds a label/value pair to the header to be sent

AJAX

XMLHttpRequest()



```
var XMLHttpRequest = new XMLHttpRequest();
XMLHttpRequest.onreadystatechange = function() {
    if (XMLHttpRequest.readyState == XMLHttpRequest.DONE)
    { console.log(XMLHttpRequest.responseText); }
}
XMLHttpRequest.open('GET', 'http://google.com', true);
XMLHttpRequest.send(); // Alte Browser brauchen null als Arg
```


AJAX

Austausch zwischen Server und Client meist über festgelegte Datenformate

- XML (Extensible Markup Language)
 - > Rein textbasiert, keine echte Typisierung
 - > Teilweise unklar ob Sachen ein Attribut oder Tag sein sollten
 - > Selbstbeschreibend, von daher generell viel Overhead
 - Schließende Tags
 - Arrays

- JavaScript Object Notation (JSON)
 - > Typisierung (jedoch mit wenig Datentypen)
 - > Keine Unterscheidung in Attribut und Tag
 - > Einfach zu lesen
 - > Kompaktheit, von daher wenig Overhead
 - > Besser für den Austausch in AJAX geeignet

Vorteile

- Keine Auswirkungen auf die Darstellung der Seite
- Verringerte Serverlast
- Erhöhte Benutzerfreundlichkeit
 - > Beispiel: Anzeigen eines Ladeindikators möglich

Nachteile

- Bruch mit klassischen Technologien
 - > Zurück-Button des Browser funktioniert nicht
 - > Komplexität der URL-Ressource hoch, ggf. fehlende Eindeutigkeit (von daher auch Probleme mit Bookmarks)
 - > Benutzerempfinden bzgl. der Rückmeldungen hängt stark von der Programmierung ab
 - > „Suchmaschinenlesbarkeit“ bzw. Zugriff ohne JavaScript

Hinweis:

- Die bisher ausgeführte Methodik über XMLHttpRequest funktioniert zwar bestens, jedoch birgt sie das Problem der Callback-Hell, einer der Gründe für das Einführen von Promises und async-await
- Mit der Callback-Hell wird das Problem bezeichnet, dass Funktionsaufrufe auch aufgrund der Callbacks tief ineinander geschachtelt sind, was zu unübersichtlichen und fehlerträchtigem Code führt
- **Gibt es hier keine moderne Art über Promises?**

Modernere Schnittstelle auf Basis von Promises

Gilt neue Schnittstelle zur Nutzung von AJAX

GET-POST-Interaktion eher mit Fetch, DOM-Manipulation später mit querySelector

```
fetch(url)
  // optionale Header-Felder { method : 'post' , ...}
  .then(function() { // meistens mit resp-Objekt
    // Hier verarbeitet man die Response
  })
  .catch(function() {
    // Fehlerhandlung
  });
```

Quelle: <https://scotch.io/tutorials/how-to-use-the-javascript-fetch-api-to-get-data>

Fetch-API

Die lange aber unschöne Version

```
fetch('./api/some.json')
  .then(
    function(response) {
      if (response.status !== 200) {
        console.log(Houston, wir haben ein Problem. Status Code: ' +
          response.status);
        return;
      }
      // Wir erhalten hier ein JSON-Objekt im Body
      // das wir zu einem nativen JS-Objekt machen
      response.json().then(data => console.log(data));
    }
  )
  .catch(function(err) {
    console.log('Fetch Error :-S', err);
  });
```

Quelle: <https://developers.google.com/web/updates/2015/03/introduction-to-fetch>

Fetch-API

Die kurze then-Kaskade, besser aber nicht super

```
fetch(url) // Lade das Objekt (JSON-Format)
.then((resp) => resp.json()) // Jetzt haben wir ein Objekt
.then(function(obj) {
    let returned_object = obj; // Verarbeite das Objekt
    ...
})
.catch(function() {
    // Fehlerbehandlung
});
```

Quelle: <https://scotch.io/tutorials/how-to-use-the-javascript-fetch-api-to-get-data>

Fetch-API

Merke



```
response.json().then(data => console.log(data)) ;
```

Response.json und response.text liefern ein Promise!

```
1 response.json().then(data => {  
2   // do something with your data  
3 });
```

Fetch-API

das response-Objekt

- `response.json()` returniert über eine resolved-Promise ein JSON-Objekt
- `response.text()` returniert über eine resolved-Promise einfachen Text
- `response.formData()` returniert über eine resolved-Promise FormData (Key-Value Paare)
- `response.blob()` returniert über eine resolved-Promise einen BLOB (nicht änderbarer File-ähnlicher Stream)
- `response.arrayBuffer()` returniert über eine resolved-Promise ein Array Puffer binäre Daten fester Länge)

Fetch-API mit POST

```
const url = ...;
// Die von uns versendeten Daten (serialisiert über JSON)
let data = {
    name: 'Volker Sander'
}
// POST, Content-Type und body setzen
let fetchData = {
    method: 'POST',
    body: data,
    headers: { 'Content-Type': 'application/json' }
}
fetch(url, fetchData)
    .then(res =>
        console.log('Objekt übertragen, Response ist: ', res);
    );
```

Quelle: <https://scotch.io/tutorials/how-to-use-the-javascript-fetch-api-to-get-data>

AJAX

FETCH vs XMLHttpRequest()

```
var XMLHttpRequest = new XMLHttpRequest();
XMLHttpRequest.onreadystatechange = function() {
    if (XMLHttpRequest.readyState == XMLHttpRequest.DONE)
    { console.log(XMLHttpRequest.responseText); }
}
XMLHttpRequest.open('GET', 'http://google.com', true);
XMLHttpRequest.send(null);
```

```
fetch('http://google.com')
.then((resp) => resp.text()) // Transform the data into text
.then(data => console.log(data.results)); // Get the results
.catch(error => console.error('error:', error));
```

FETCH geht noch schöner!

Async await!!!

```
async function getJson(url)
{
    let response = await fetch(url)
    if (response.ok)
        return await response.json()
    throw new Error(`HTTP error! status: ${response.status}`)
}
```

response.json liefert
ein promise zurück,
daher then oder
besser await

```
try {
    const receivedObject =
        (async() => { await getJson("https://abc.org/") })()
    console.log(receivedObject)
    const str = JSON.stringify(receivedObject)
    const str = JSON.parse(str)
}
catch(err => console.log(err))
```

Empfehlung:

- Die Nutzung von async-await macht FETCH deutlich übersichtlicher
- Sie haben auch keine Callback-Hell
- Sie können await aber nur sicher in einer async-Funktion verwenden

```
const fetchWhatever = async () => {  
    const resp = await fetch('url') // ggf. throw  
    let users = await resp.json()  
}  
  
(async () => {  
    await fetchWhatever()  
    ...  
})()
```

oder `fetchWhatever().then(...)`

oder `fetchWhatever()` **Keine Kontrolle darüber was dann gemacht werden soll**

Fetch mit Chaining

```
function loadJson(url) {
  return fetch(url)
    .then(response => response.json());
}

function loadGithubUser(name) {
  return loadJson(`https://api.github.com/users/${name}`);
}

function showAvatar(githubUser) {
  return new Promise(function(resolve, reject) {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser);
    }, 3000);
  });
}

// Use them:
loadJson('/article/promise-chaining/user.json')
  .then(user => loadGithubUser(user.name))
  .then(showAvatar)
  .then(githubUser => alert(`Finished showing ${githubUser.name}`));
// ...
```

<https://javascript.info/promise-chaining>

Fetch mit await

```
async function showAvatar() {  
  
    // read our JSON  
    let response = await fetch('/article/promise-chaining/user.json');  
    let user = await response.json();  
  
    // read github user  
    let githubResponse = await fetch(`https://api.github.com/users/${user.name}`);  
    let githubUser = await githubResponse.json();  
  
    // show the avatar  
    let img = document.createElement('img');  
    img.src = githubUser.avatar_url;  
    img.className = "promise-avatar-example";  
    document.body.append(img);  
  
    // wait 3 seconds  
    await new Promise((resolve, reject) => setTimeout(resolve, 3000));  
  
    img.remove();  
  
    return githubUser;  
}  
  
showAvatar();
```

<https://javascript.info/promise-chaining>

Top-Level await in Modulen (exports)

IE	Edge *	Firefox	Chrome	Safari	Opera	Safari on iOS *	Opera Mini *	Android Browser *	Opera Mobile *
					10-72				
	12-88	2-88	4-88	3.1-14.1	73-74	3.2-14.8			
6-10	89-99	89-98	89-99	15-15.3	75-82	15-15.3		2.1-4.4.4	12-12.1
11	100	99	100	15.4	83	15.4	all	100	64
		100-101	101-103	TP					

<https://caniuse.com/?search=top%20level%20await>

FETCH eines HTML-Dokuments

Wir erhalten bei FETCH kein Document-Object!

```
// newEL sei ein Objekt zum Einfügen in den DOM-Baum
// für den einen CSS-Selektor

const fetchWhatever = async () => {
    const resp = await fetch('url') // ggf. try-catch
    return await resp.text()
}

(async () => {
    text = await fetchWhatever()
    const parser = new DOMParser()
    const document = parser.parseFromString(text,
        "text/html")
    const poi = document.querySelector(selector)
    if (poi.parentNode) {
        poi.parentNode.insertBefore(newEl,
            poi.nextSibling);
    }
})();
```


ETCH eines HTML-Dokuments

Wir erhalten bei FETCH kein Document-Object

```
const fetchWhatever = async () => {
  const resp = await fetch('url') // ggf. try-catch
  return await resp.text()
}

(async () => {
  html = await fetchWhatever()
  const parser = new DOMParser()
  const htmlDoc = parser.parseFromString(html, "text/html")
  console.log(htmlDoc.querySelector('span').textContent)
})();
```

Hinweise:

- ggf. werden mehrere Knoten des DOM-Baums spezifiziert. In diesem Fall sollte `querySelectorAll` verwendet werden.
- **Die `querySelector`-Routinen sind relativ langsam.** Alternativ wären die
`document.getElementById` `document.getElementsByClassName`
`document.getElementsByTagName`

Der Zugriff auf das Document-Objekt wird beim klassischen Laden von HTML-Dokumenten durch den Browser gewährleistet

Wir können bei klassischen Single-Page-Anwendungen von einem vorhandenen Zugriff auf das Document-Objekt ausgehen

Hier laden wir Informationen nach und integrieren diese ggf. in das existierende Document-Object

Wenn aber mit FETCH ein neues DOM geladen wird, müssen wir den Umweg gehen

DOM-Manipulation

Die folgenden Methoden bieten sich für das document-Objekt an

- `querySelector(selectors)` liefert das erste Element des Dokuments zurück, welches die/den Selektor erfüllt, z.B. `‘.meineKlasse‘`
- `querySelectorAll(selectors)` liefert alle Elemente des Dokuments zurück, wobei hier auch eine Liste von Selektoren verwendet werden kann, `"#id1, #id2, #id3, #id4"`
- `getElementById(id)` schnelle Methode die ein einzelnes Element zurückliefert mit der Id
- `getElementsByClassName(class)` schnelle Methode die alle Elemente zurückliefert mit den angegebenen Klassenauszeichnungen (`"class1 class2 class3"`)
- `getElementsByTagName(tag)` schnelle Methode die alle Elemente zurückliefert für das angegebene einzelne Auszeichnungselement

DOM-Manipulation

Jedes so erhaltene Element kann entweder mit Events versehen werden oder dient als Pointer zur Manipulation des DOM-Baums. Wir gehen jetzt davon aus, dass wir ein solches Element haben und nutzen entsprechende Methoden

- `addEventListener(type, listener)` bindet den Event-Typ an das Element mit dem angegebenen Listener, z.B. 'submit' für Formulare, 'click' bei Mausbetätigung
<https://developer.mozilla.org/de/docs/Web/Events>
- `classList` gibt die Liste der (css-)Klassen des Elements zurück, die dann erweitert/geändert werden kann
- `appendChild(node)`, `insertBefore(newNode, referenceNode)` die Elemente werden mit `document.createElement('name')` erzeugt. Dieses muss noch `append` o.ä. zum `document` hinzugefügt werden.