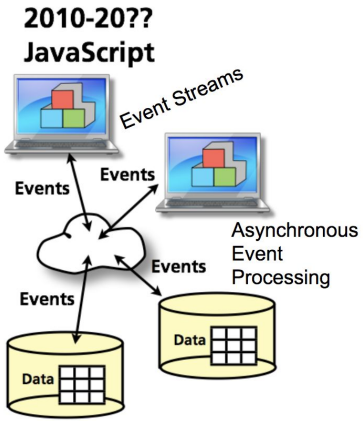
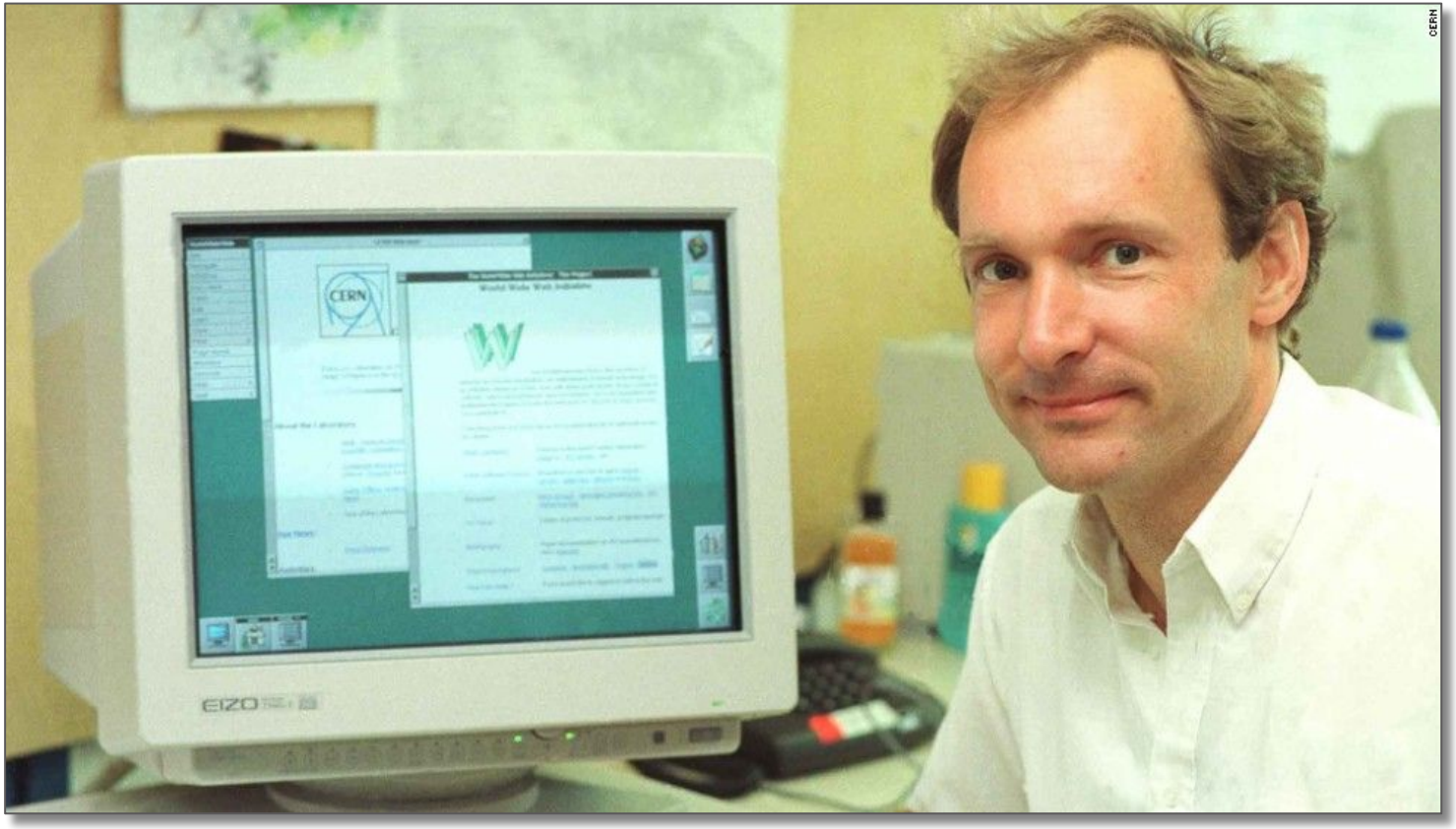
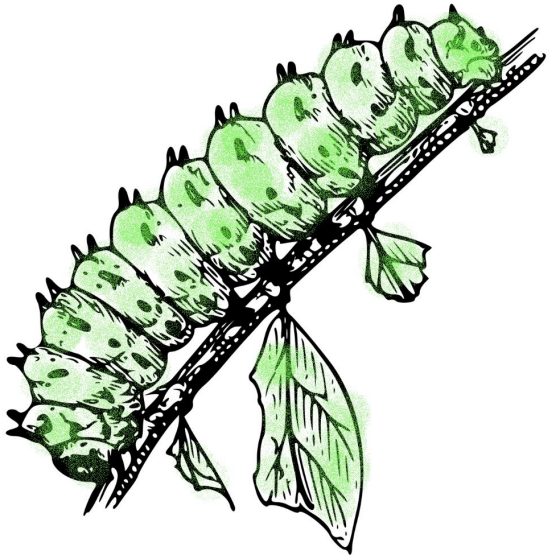
Asynchrones JavaScript



Was sind dafür die besten Programmiersprachen-Konstrukte?

Wdh.: Was war die primäre Absicht des TBL?





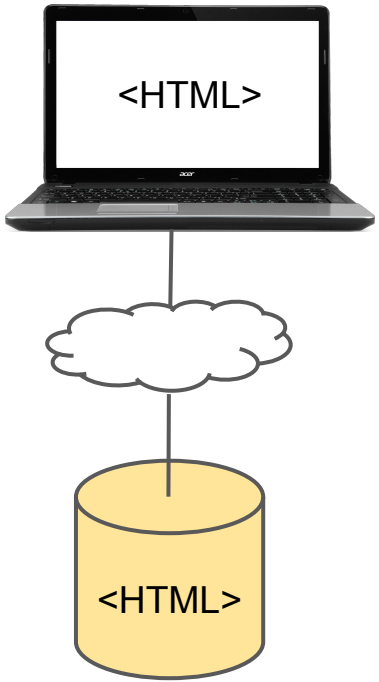
Metamorphose des WWW

Free Digital Library for everyone

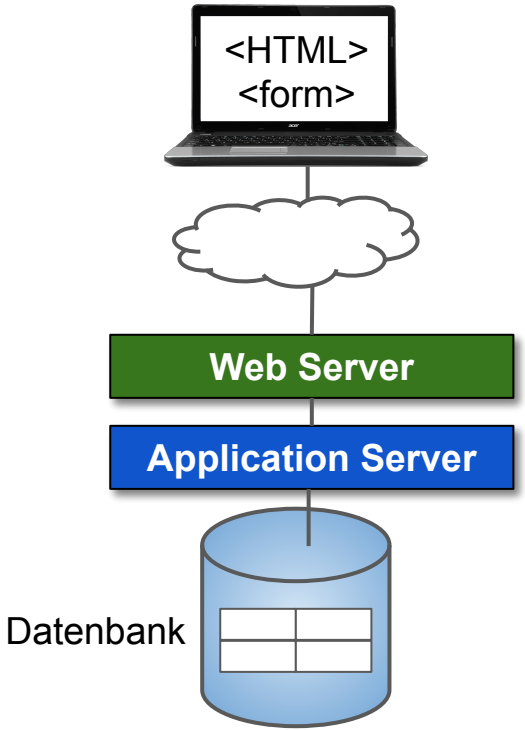
Das zentrale Nervensystem der Gesellschaft

Paradigmen des WWW

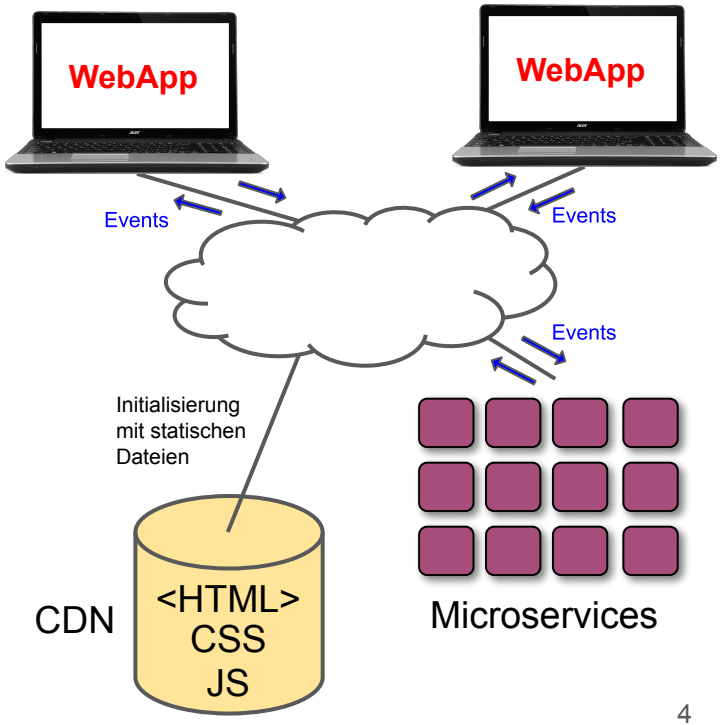
bis 1997
statisches HTML
Paradigma: **Digital Library**



ab 1997
Paradigmen:
Dokumentverarbeitung
Formulare



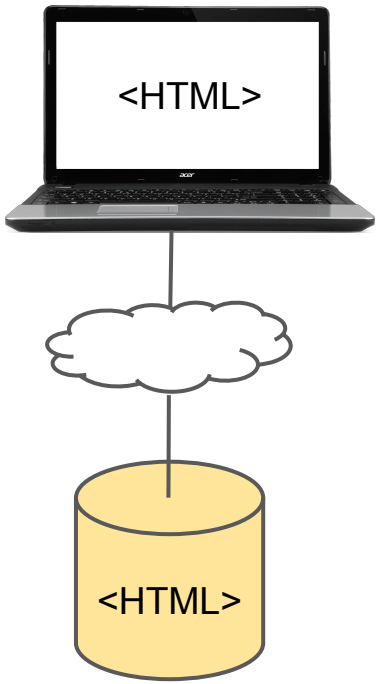
heute
Paradigma: **Das Nervensystem der Gesellschaft**



http://gigaom.com/cloud/node-js-and-the-javascript-age/

Metamorphose des WWW

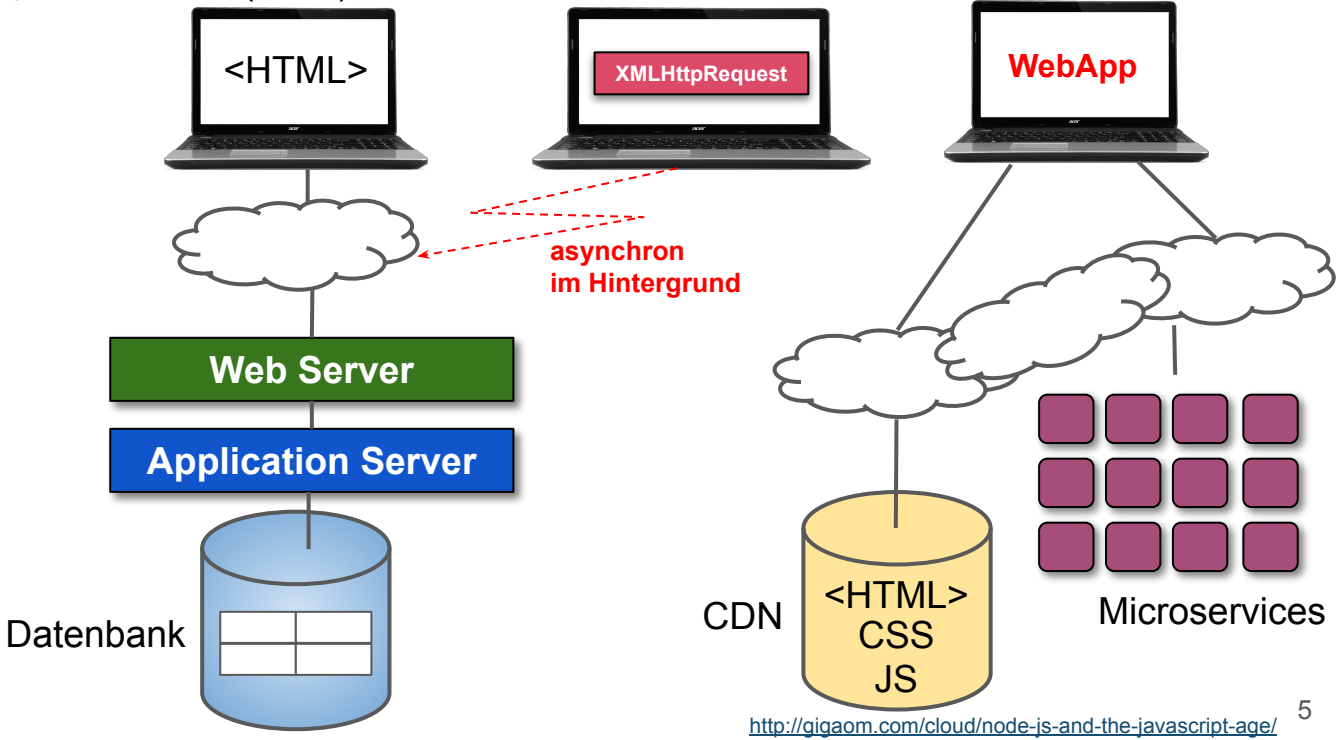
bis 1997
statisches HTML
Paradigma: **Digital Library**



ab 1997
Server-side Applications
mit >3-Schichten-Architektur
(MVC)

ab 2000
AJAX

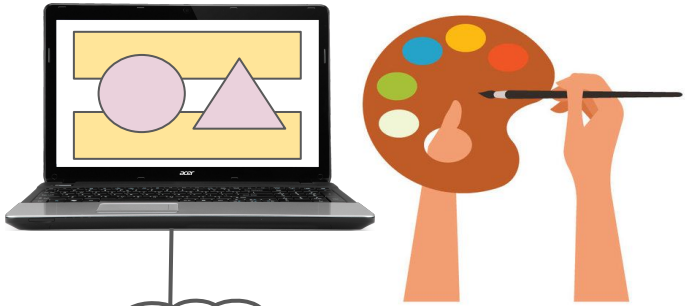
heute
Client-side Applications
mit Microservices



http://gigaom.com/cloud/node-js-and-the-javascript-age/

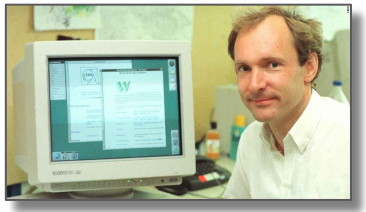
Der Kategorienfehler des WWW

Digital Media Library



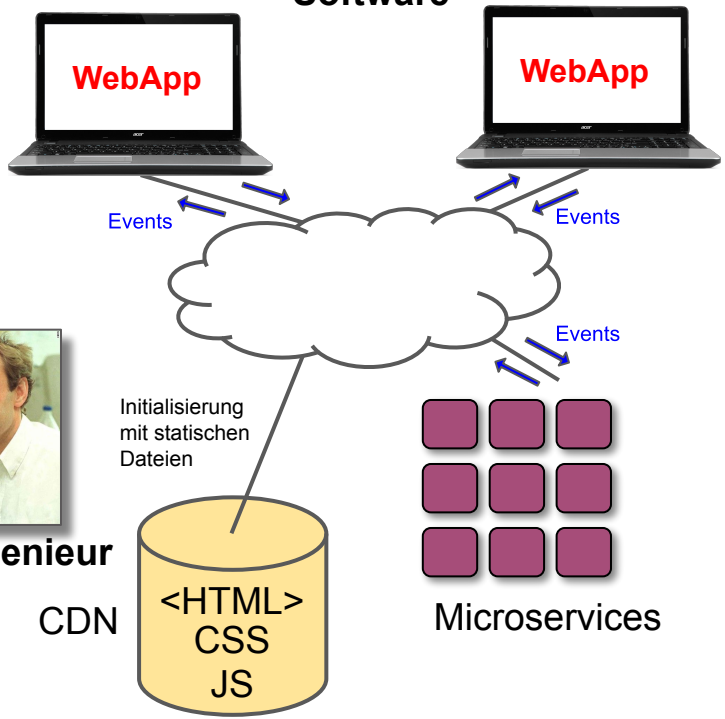
<HTML>

Beruf: Medien Designer



Beruf: Software-Ingenieur

Verteilte, Event-getriebene Software



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

Was heißt "Event-getrieben"?
⇒ Geschichte der Parallelität in CPU & OS

Jahr	Prinzip	OS-Prinzip	Beispiele
1990	1 CPU = 1 Prozess	**Single Process**	DOS
1995	1 CPU + viele Prozesse	**Cooperative Multitasking**	Windows 95, 98
1996	1 CPU + viele **Prozesse**	**Preemptive Multitasking**	Windows NT, 2000, Linux
2000	2++ CPUs + viele Prozesse + viele **Threads**	**Symmetric MultiThreading (SMT)**	Pentium 4 (Intel): Intel **Hyper-Threading**: New CPU instruction code for parallelization

Was heißt "Event-getrieben"?

Processes	Threads	Events
Top-level execution container	runs inside a process	runs inside a thread
separate memory space	shared memory space	same memory
Communicate via Inter-Process Communication (IPC)	Many communication options, Synchronisierung auf gemeinsamen Speicher notwendig, Mutex, Semaphoren, Critical Region, Rendezvous, Monitore, Warteschlangen, ...	**keine Synchronisierung erforderlich**

Event-getriebene Web-Server

LAMP	Java EE	Node.js
multi-process	multi-threaded	single-threaded
1 process per request	1 thread per request	1 event per request
PHP interpretiert	compiliert	JS interpretiert
blocking I/O process switching	blocking I/O thread switching	non-blocking I/O event-driven
beliebtester WWW-Server	Unternehmen	stark wachsend
Verschiedene Experten für Frontend und Backend	Verschiedene Experten für Frontend und Backend	JS-Programmierer für beide Seiten, Client & Server

JavaScript is single threaded:

- Keine Nebenläufigkeitsartefakte
- Run-to-completion semantics

```
1  function A(){
2    setTimeout( function B(){
3          console.log('Second');
4        }, 0 );
5    console.log('First');
6  }
7
8  A();
```

Minimale Zeit, nicht garantierte Zeit

- JavaScript-Funktionen werden immer Zeile für Zeile ungestört vollständig bis zu Ende ausgeführt
  - keine Parallelität
  - keine Nebenläufigkeitsartefakte
  - kein vorzeitiger Abbruch
- In Zeile 2 wird B in die Timer Queue gelegt, aber noch nicht gestartet
- Erst beim nächsten Durchlauf der Event Loop kommt B an die Reihe
- Es wird immer nur 1 JavaScript-Funktion ausgeführt

https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop#Run-to-completion

Gliederung

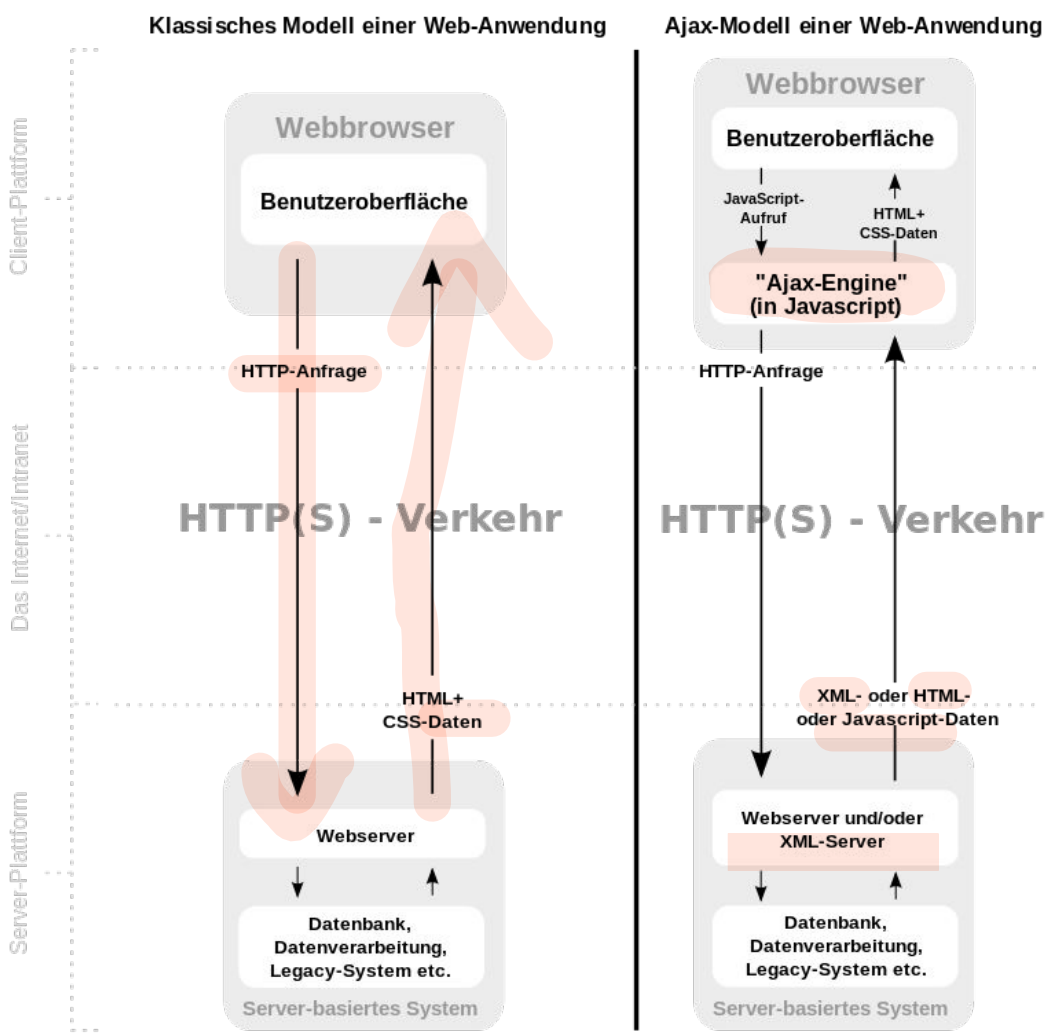
JavaScript-Zeitalter ✓

1. AJAX
2. XMLHttpRequest
3. Promise
4. fetch-API
5. async / await
6. Event Loop
7. Events
8. Worker-API

1. AJAX

- AJAX = "Asynchronous JavaScript and XML"
- Begriff def. von Jesse James Garret 2005

- Neuer Interaktionsstil:
  - Update ohne Page Reload



https://de.wikipedia.org/wiki/Ajax_(Programmierung)

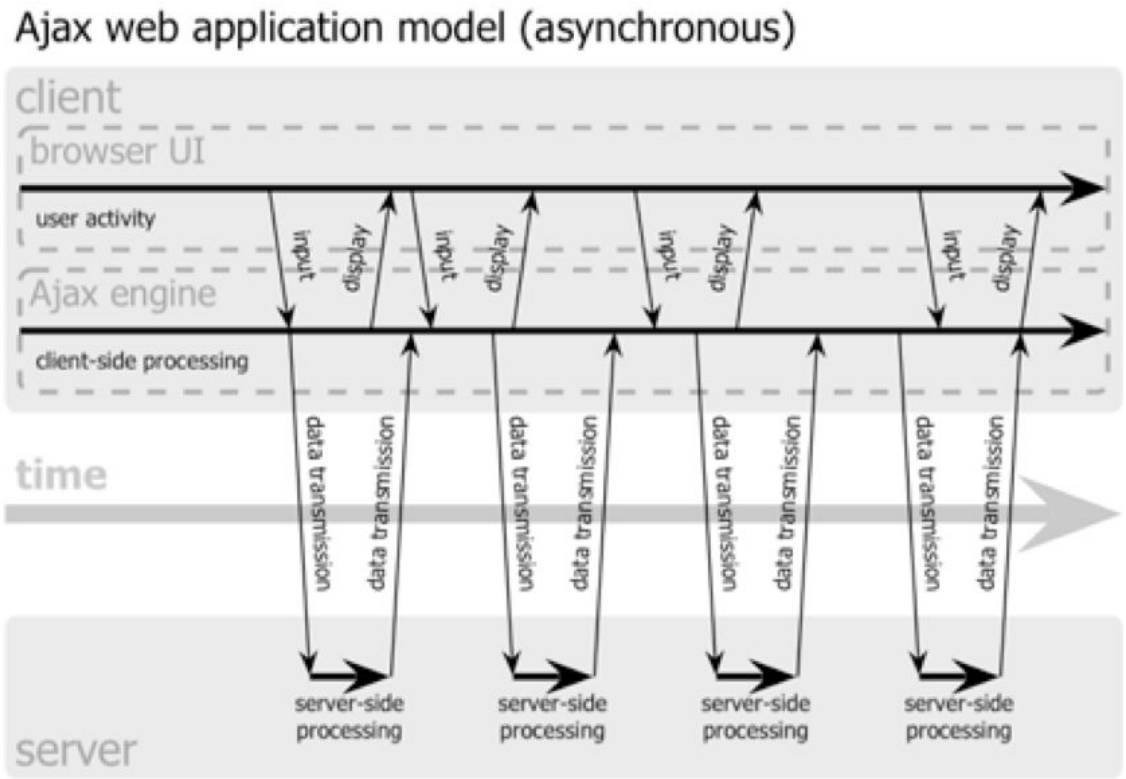
WWW wird asynchron

WWW war synchron:

- GET-Request
- Page Load

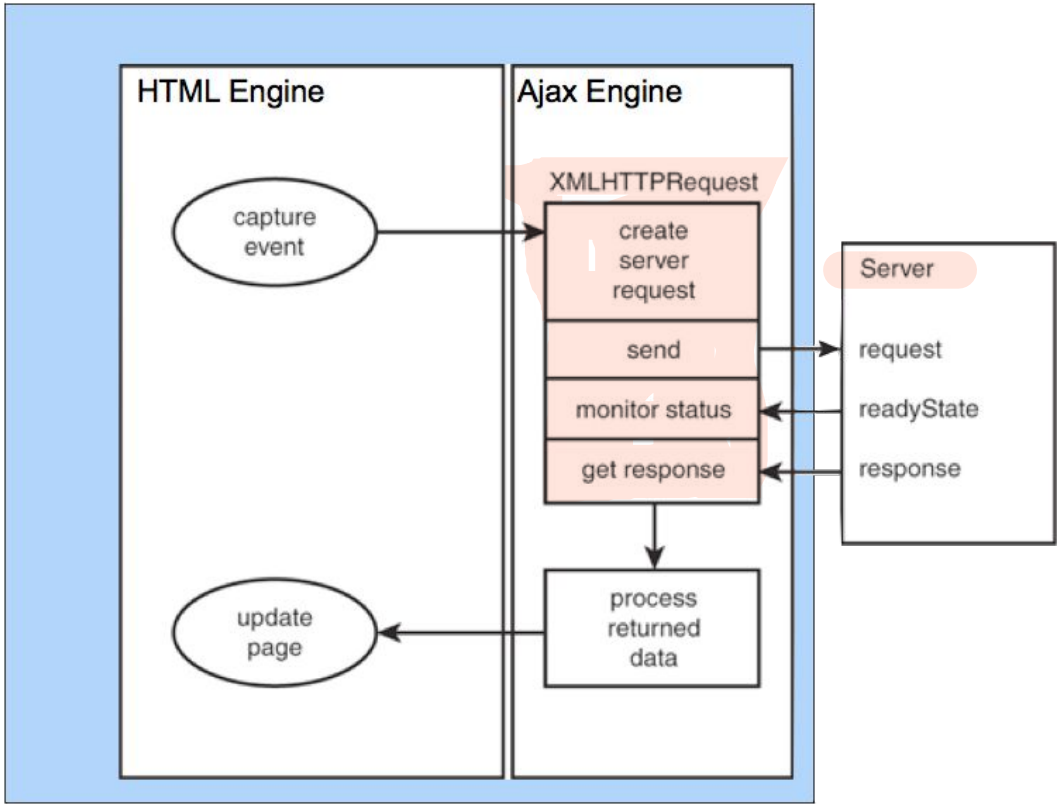
⇒ Problem:
Browser blockiert

Durch Ajax Page Refresh asynchron



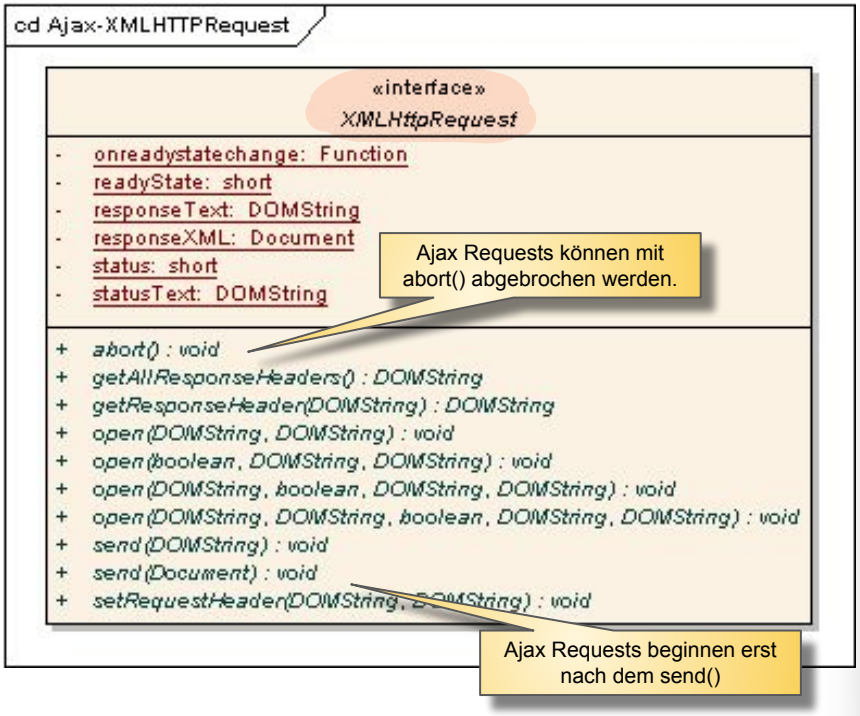
Quelle Jesse James Garrett 2005
http://www.adaptivepath.com/publications/essays/archives/000385.php

new XMLHttpRequest() ist die Ajax Engine

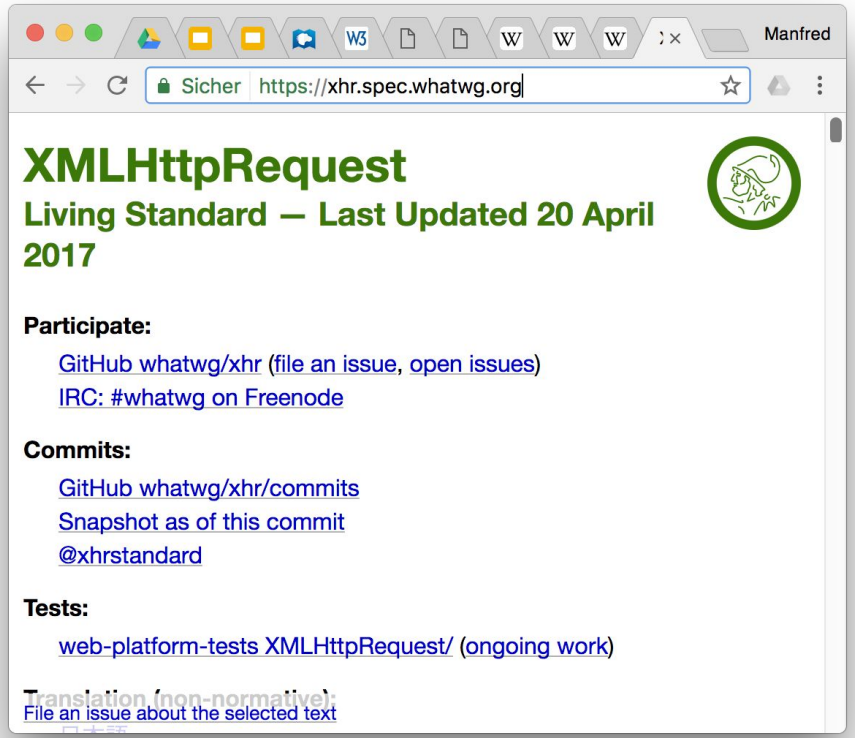


- XMLHttpRequest ist das Interface des Browsers, das die AJAX-API zur Verfügung stellt
- Aufruf:
  ```
  var xhr = new XMLHttpRequest();
  ```
- synchron / asynchron
- nicht nur XML

Spezifikation durch WHATWG



Quelle: http://de.wikipedia.org/wiki/XMLHttpRequest



https://xhr.spec.whatwg.org/#states

"xhr.readyState === 4" heißt "fertig"

4.4. States

For web developers (non-normative)

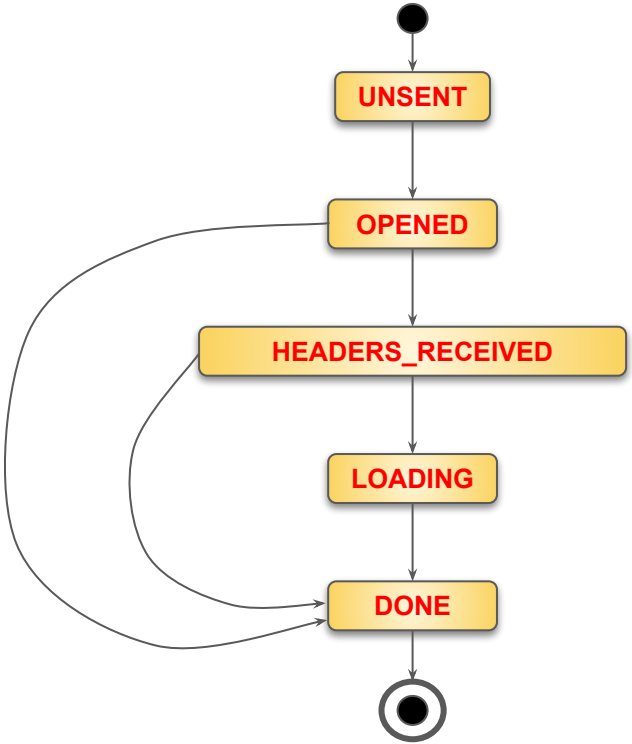
***client* . readyState**

Returns *client*'s state.

The **readyState** attribute's getter must return the value from the table below in the cell of the second column, from the row where the value in the cell in the first column is context object's state:

unsent	**UNSENT** (numeric value 0)	The object has been constructed.
opened	**OPENED** (numeric value 1)	The `open()` method has been successfully invoked. During this state request headers can be set using `setRequestHeader()` and the fetch can be initiated using the `send()` method.
headers received	**HEADERS_RECEIVED** (numeric value 2)	All redirects (if any) have been followed and all HTTP headers of the response have been received.
loading	**LOADING** (numeric value 3)	The response's body is being received.
done	**DONE** (numeric value 4)	The data transfer has been completed or something went wrong during the transfer (e.g. infinite redirects).

Lebenszyklus eines AJAX-Requests



https://xhr.spec.whatwg.org/#states

Neues HTML im Hintergrund zusätzlich laden

```
<h1>Ajax</h1>
<button>Make a request</button>
<div id="result"></div>
<script>
  const xhr = new XMLHttpRequest();
  const div = document.querySelector("#result");
  document.querySelector("button").onclick = makeRequest;

  function makeRequest() {
    xhr.onreadystatechange = alertContents;
    xhr.open('GET', './test.html');
    xhr.send();
  }
```

```
function alertContents() {
    if (xhr.readyState === XMLHttpRequest.DONE) {
      if (xhr.status === 200) {
        div.innerHTML = xhr.responseText;
      } else {
        div.innerHTML = '<h2>There was a problem.</h2>';
      }
    }
  }
</script>
```

xhr.readyState === 4

HTTP Return Code

https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX/Getting_Started

JSON-Daten im Hintergrund laden und anzeigen

```
<h1>JSON Demo</h1>
<button>Make a request</button>
<div id="result"></div>
<script>
 const xhr = new XMLHttpRequest();
 const div = document.querySelector("#result");
 document.querySelector("button").onclick =
                              makeRequest( './data.json' );

 function makeRequest( url ) {
  return function(){
    xhr.onreadystatechange = alertContents( url );
    xhr.open( 'GET', url );
    xhr.send();
  }
 }
```

```
class Person {
  constructor( json ){
    Object.entries( json ).forEach( ([key, value]) => {
      this[key] = value;
    } );
  }
  toHTML(){
    let out = '<h1>Person</h1>';
    Object.entries( this ).forEach( ([key, value]) => {
      out += `<b>${key}:</b> ${value}<br>`;
    } );
    return out;
  }
 }
</script>
```

```
function alertContents( url ) {
  return function(){
    if (xhr.readyState === XMLHttpRequest.DONE) {
      if (xhr.status === 200) {
        const text = xhr.responseText;
        const json = JSON.parse( text );
        const person = new Person( json );
        div.innerHTML = person.toHTML();
      } else {
        div.innerHTML = `<b>${xhr.status}:</b>
                              ${url} ${xhr.statusText}<br>`;
      }
    }
  }
}
```

JSON Demo

Make a request

Person

name: Martin Mustermann
address: Hochschule Bonn-Rhein-Sieg, Sankt Augustin

https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX/Getting_Started

Site Speed, SEO und Absprungrate

SEO = Search Engine Optimization

Google Webmaster Central Blog

Official news on crawling and indexing sites for the Google index

Using site speed in web search ranking

Friday, April 09, 2010

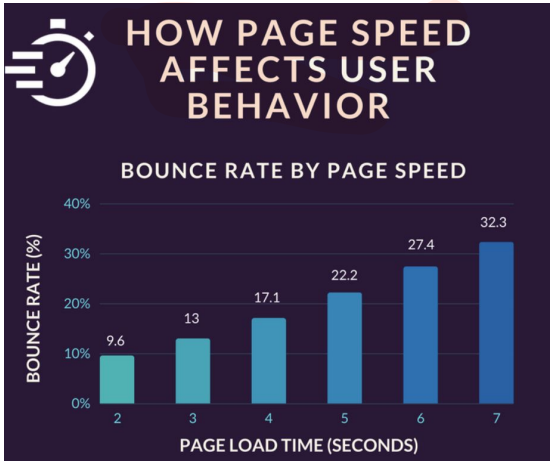
Webmaster Level: All

You may have heard that here at Google we're obsessed with speed, in our products and on the web. As part of that effort, today we're including a new signal in our search ranking algorithms: site speed. Site speed reflects how quickly a website responds to web requests.

https://webmasters.googleblog.com/2010/04/using-site-speed-in-web-search-ranking.html

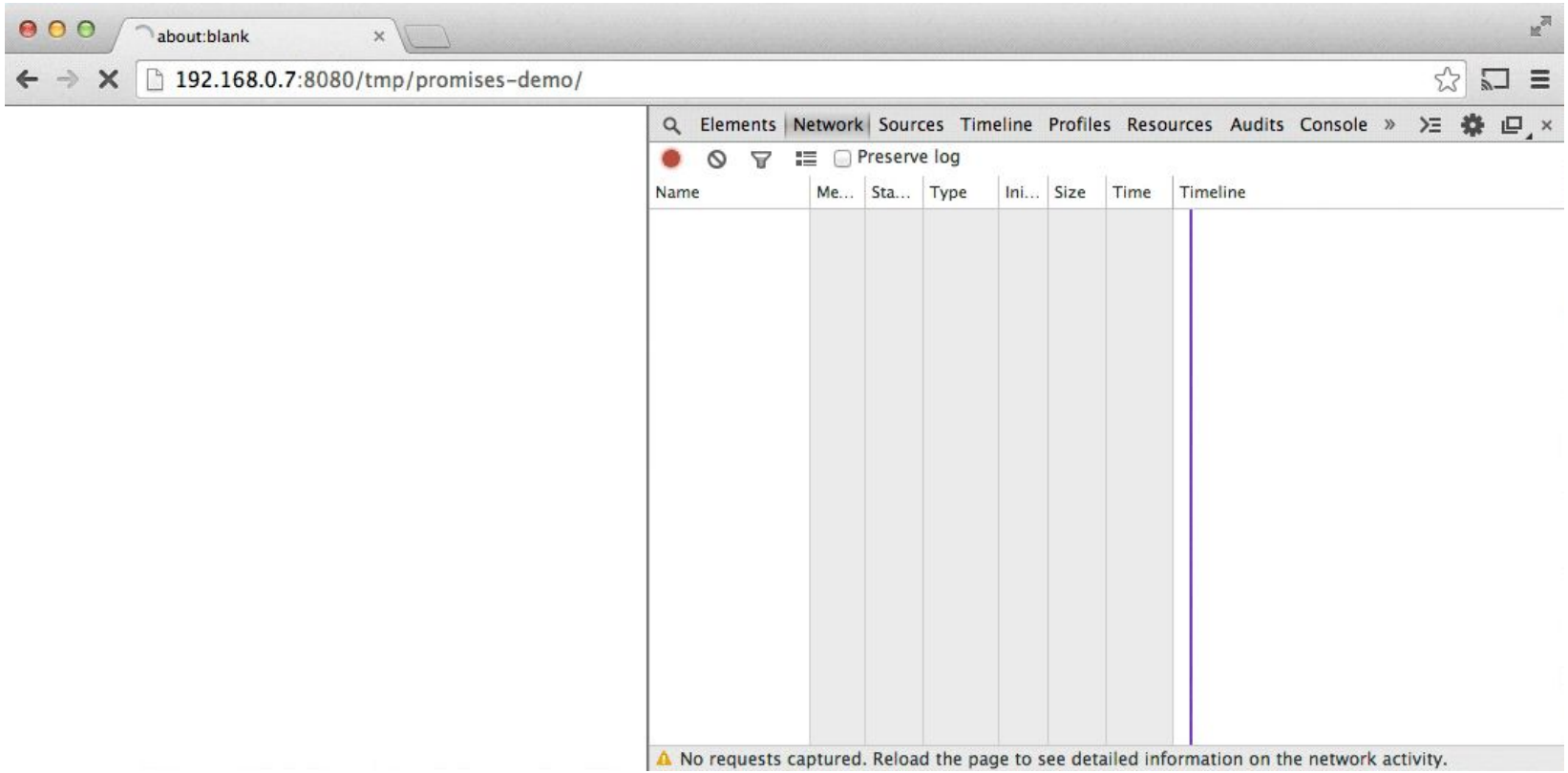
"We encourage you to start looking at your site's speed — not only to improve your ranking in search engines, but also to improve everyone's experience on the Internet."

Google Search Quality Team



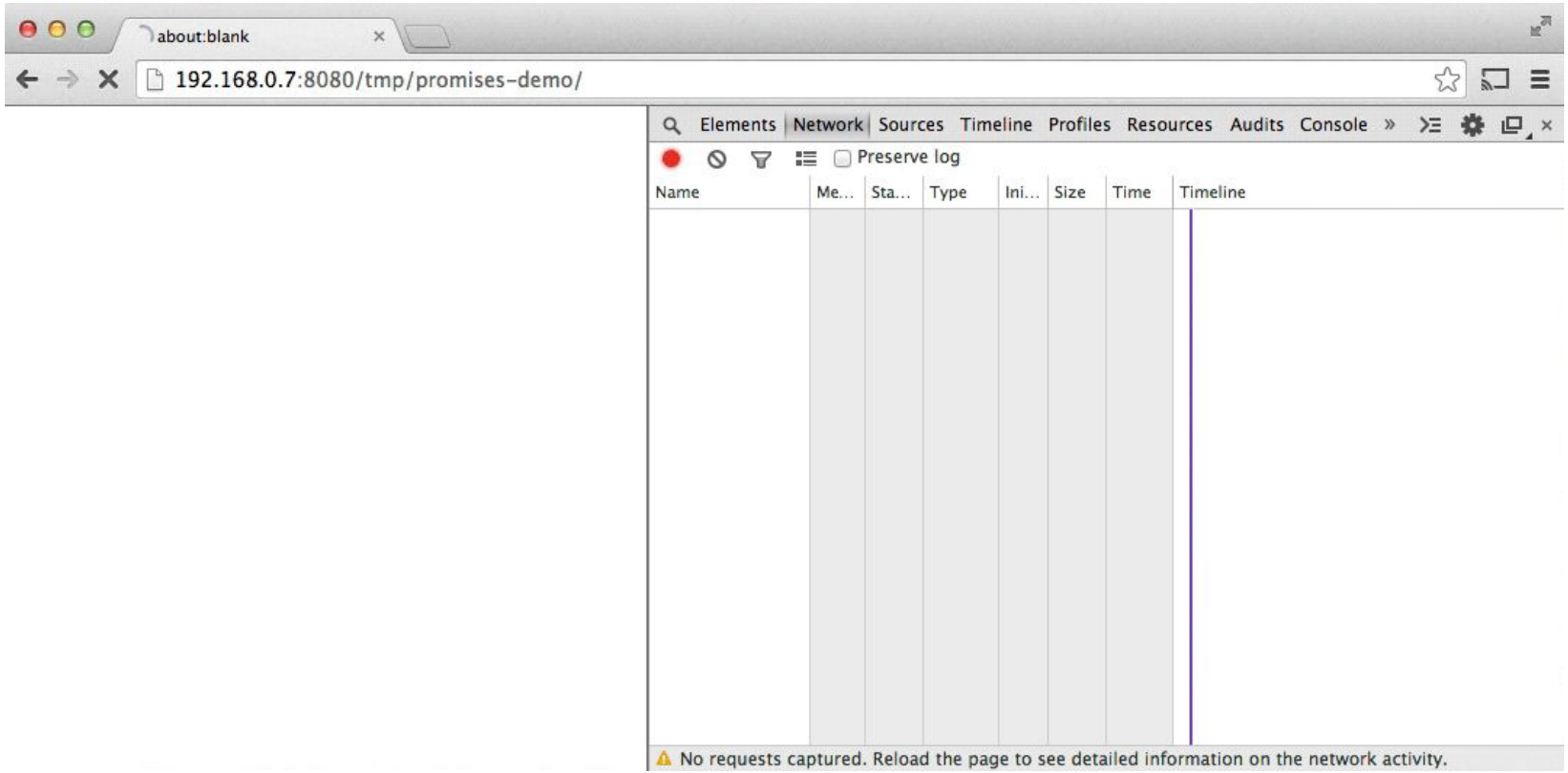
https://www.section.io/blog/page-load-time-bounce-rate/

Laden seriell



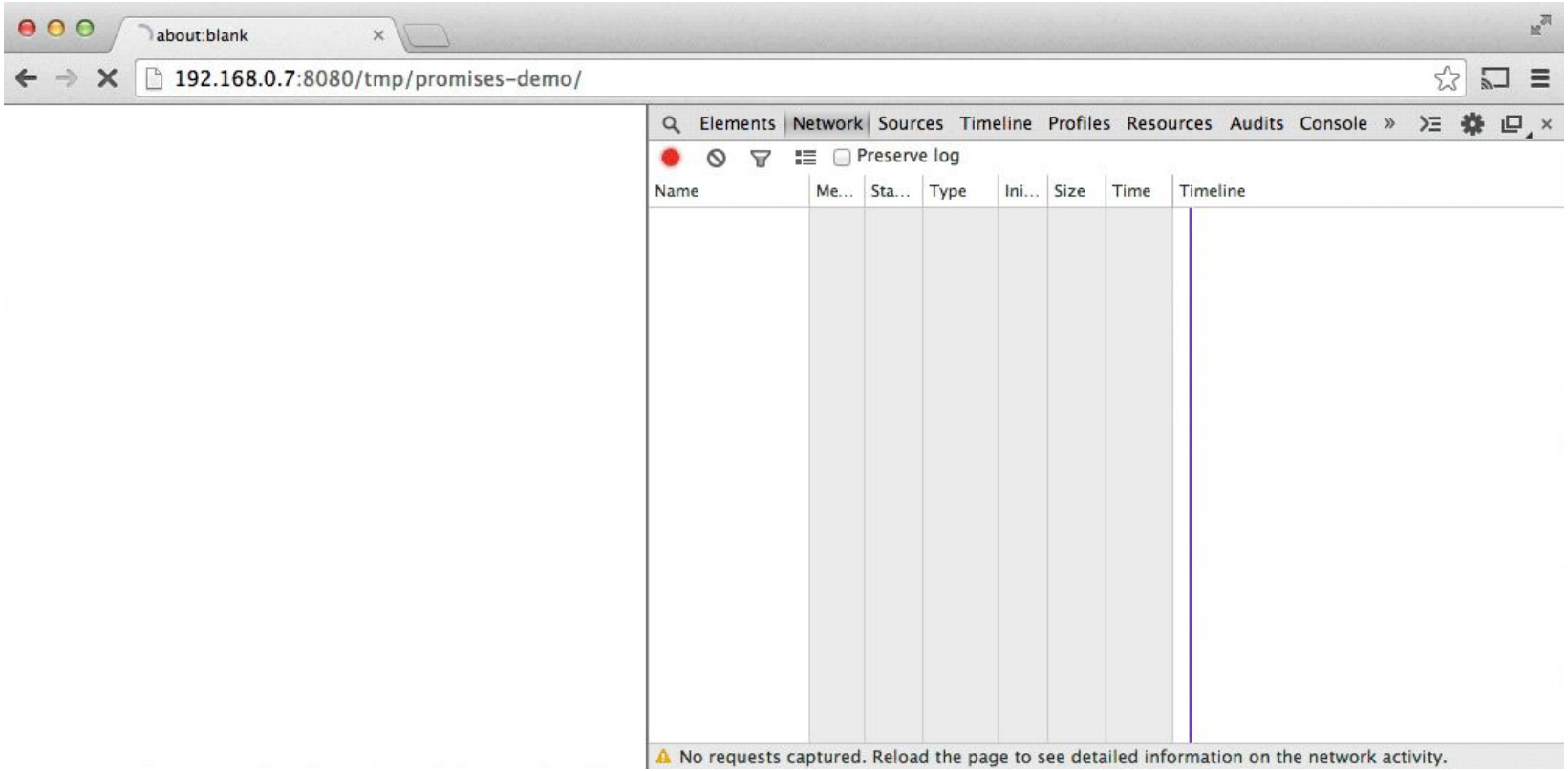
https://developers.google.com/web/fundamentals/primers/imgs/promise1.gif

Laden parallel und alle zusammen anzeigen



https://developers.google.com/web/fundamentals/primers/imgs/promise1.gif

So schnell wie möglich anzeigen & parallel



https://developers.google.com/web/fundamentals/primers/imgs/promise1.gif

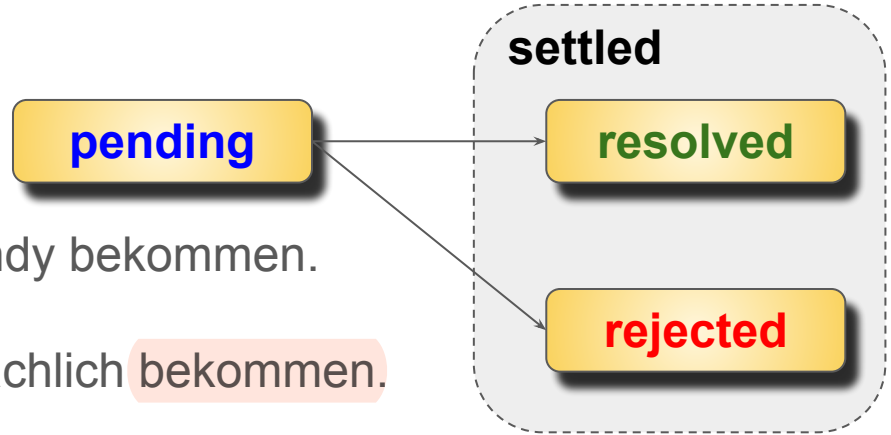
Gliederung

1. AJAX
2. XMLHttpRequest
3. Promise
4. fetch-API
5. async / await
6. Event Loop
7. Events
8. Worker-API

Promises

Idee zu Promise: "Ihre Eltern versprechen Ihnen ein neues Handy."

3 Zustände:



- Promise is **pending**:
  Sie wissen nicht, ob Sie das Handy bekommen.
- Promise is **resolved**:
  Sie haben das neue Handy tatsächlich bekommen.
- Promise is **rejected**:
  Sie wissen, dass Sie das neue Handy nicht bekommen.
- Promise is **settled** = resolved or rejected

Promise erzeugen & Promise konsumieren

```
const prom = new Promise(
    function (resolve, reject) {
        …
        if ( successful ) {
            resolve( value ); // success
        } else {
            reject( error ); // failure
        }
    });
```

Kurzformen:

```
Promise.resolve()
Promise.reject()
```

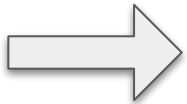
```
prom
.then( value => { /* fulfillment */ })
.catch( error => { /* rejection */ })
.finally( value => { /* finalisation */ });
```

```
const prom1 = new Promise(resolve => {
 setTimeout(_=> resolve("Success!"), 250);
});

const prom2 = new Promise(( resolve, reject ) => {
 setTimeout(_=> resolve("Success!"), 250);
 setTimeout(_=> reject("Denied!"), 100);
});

prom1.then( console.log );
prom2.then( console.log ).catch( console.log );

Promise.resolve(123).then(( x ) => console.log( x ));
Promise.reject(456).catch( console.log );
```

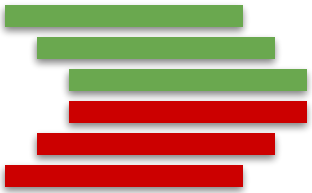
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/promise

Promises als Alternative zu Callbacks

```
setTimeout( () => {
 console.log( 1 );
 setTimeout( () => {
   console.log( 2 );
   setTimeout( () => {
   console.log( 3 );
   }, 3000 );
 }, 2000 );
}, 1000 );
```



```
function wait( millisec ){
 return new Promise( resolve => setTimeout( resolve, millisec ) );
}

wait( 1000 )
 .then( _=> { console.log( 1 ); return wait( 2000 ) } )
 .then( _=> { console.log( 2 ); return wait( 3000 ) } )
 .then( _=>   console.log( 3 )  );
```



Promise.then

P.**then**(onFulfilled, onRejected) ergibt wieder ein Promise

```
Promise.resolve(123)
 .then(function (value1) {
   return value1;
 })
 .then(function (value2) {
   console.log(value2); // 123
 });
```

Informationen weiterreichen

Syntax

```
p.then(onFulfilled[, onRejected]);

p.then(value => {
  // fulfillment
}, reason => {
  // rejection
});
```

```
Promise.resolve(1).then(x=>++x).then(x=>++x).then( console.log );
Promise.reject(1).catch(x=>++x).finally( console.log('fini') );
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then

Promise Chaining

By Jake Archibald
Human boy working on web standards at Google

https://developers.google.com/web/fundamentals/getting-started/primers/promises

```
Promise.resolve($.getJSON('/path/to/my/api'))
 .then(function() {
 return asyncThing2();
}).then(function() {
 return asyncThing3();
}).catch(function(err) {
 return asyncRecovery1();
}).then(function() {
 return asyncThing4();
}, function(err) {
 return asyncRecovery2();
}).catch(function(err) {
 console.log("Don't worry about it");
}).then(function() {
 console.log("All done!");
})
```

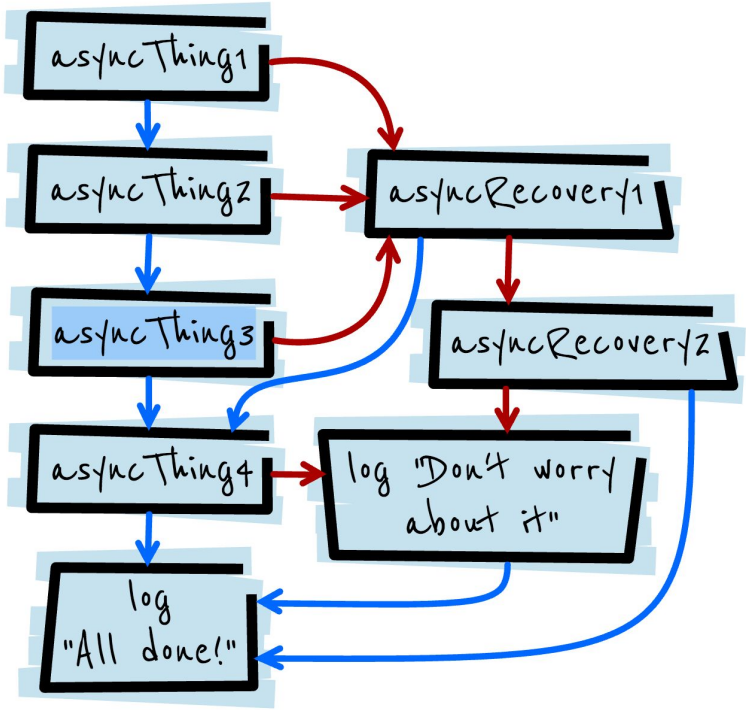
asyncThing1

Syntax

```
p.then(onFulfilled[, onRejected]);

p.then(value => {
  // fulfillment
}, reason => {
  // rejection
});
```

Versprechen gebrochen →

Versprechen erfüllt →



Mit Promise.all synchronisieren

```
function delay(ms) {
 return new Promise(function (resolve) {
   setTimeout(resolve, ms)
 });
}

Promise.all([
 delay(1000).then(()=>console.log('1 second have passed.')),
 delay(2000).then(()=>console.log('2 seconds have passed.'))
])
 .then(()=>console.log( "Both ready." ) );
```

```
1 second have passed.
2 seconds have passed.
Both ready.
```

Ü: Was ist die Ausgabe von … ?

```
Promise.all([
 Promise.resolve(1),
 Promise.reject(2),
 Promise.resolve(3),
 Promise.resolve(4)
]).then( console.log, console.log );
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all

Promise.race: Der Schnellste gewinnt

```
function delay(ms) {
 return new Promise(function (resolve) {
   setTimeout(resolve, ms)
 });
}

Promise.race([
 delay(1000).then(()=>console.log('1 second have passed.')),
 delay(2000).then(()=>console.log('2 seconds have passed.'))
])
 .then(()=>console.log( "First ready." ) );
```

```
1 second have passed.
First ready.
2 seconds have passed.
```

Ü: Was ist die Ausgabe von … ?

```
const delay = ms => new Promise( resolve =>
                        setTimeout( _=> resolve(ms), ms ) );

Promise.race([
 delay(100),
 delay(200),
 delay(300),
 delay(400),
]).then( console.log, console.log );
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/race

Promisifying XMLHttpRequest

```
function get(url) {
 // Return a new promise.
 return new Promise(function(resolve, reject) {
  // Do the usual XHR stuff
  var req = new XMLHttpRequest();
  req.open('GET', url);

  req.onload = function() {
   // This is called even on 404 etc
   // so check the status
   if (req.status === 200) {
    // Resolve the promise with the response text
    resolve(req.response);
   }
   else {
    // Otherwise reject with the status text
    // which will hopefully be a meaningful error
    reject(Error(req.statusText));
   }
  };

  // Handle network errors
  req.onerror = function() {
   reject(Error("Network Error"));
  };

  // Make the request
  req.send();
 });
}
get('story.json').then(function(response) {
 console.log("Success!", response);
}, function(error) {
 console.error("Failed!", error);
});
```

Nachteile:

- Kein **abort()** möglich
- Während der Bearbeitung keine Abfrage des Status möglich

https://developers.google.com/web/fundamentals/primers/promises

Gliederung

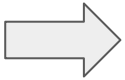
1. AJAX
2. XMLHttpRequest
3. Promise
4. fetch-API
5. async / await
6. Event Loop
7. Events
8. Worker-API

Asynchronous Reading via fetch-API

- fetch() analog zu XMLHttpRequest
- fetch() nutzt Promises statt Callbacks

XMLHttpRequest

```
var oReq = new XMLHttpRequest();
oReq.onload = reqListener;
oReq.onerror = reqError;
oReq.open('get', '/path/some.json', true);
oReq.send();

function reqListener() {
 var data = JSON.parse(this.responseText);
 console.log(data);
}

function reqError(err) {
 console.log('Fetch Error: ', err);
}
```



fetch

```
fetch('/path/some.json')
 .then( json )
 .catch(function( error ) {
   console.log('Request failed ', error);
});

function json(response) { return JSON.parse(response.json()) }
```

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

Gliederung

1. AJAX
2. XMLHttpRequest
3. Promise
4. fetch-API
5. async / await
6. Event Loop
7. Events
8. Worker-API

ES8: Neue async / await - Syntax

```
const delay = (ms, result) =>
 new Promise(resolve => setTimeout(() => resolve(result), ms));

async function delays1() {
 let a = await delay(800, "Hello, I'm in an");
 console.log(a);

 let b = await delay(400, "async function!");
 console.log(b);
}

delays1();
```

async / await

```
function delays2() {
 return delay(800, "Hello, I'm in a").then(a => {
   console.log(a);
   return delay(400, "Promise!");
 }).then(b => {
   console.log(b);
 });
}

delays2();
```

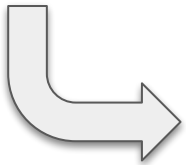
Promises

```
const delay_cb = (ms, result, cb) =>
            setTimeout(() => cb(result), ms);

function delays3() {
 delay_cb(800, "Hello, I'm in a", a => {
   console.log(a);
   delay_cb(400, "callback!", b => {
     console.log(b);
   });
 });
}

delays3();
```

Callback Hell

Rewriting a promise chain with an async function

```
function getProcessedData(url) {
  return downloadData(url) // returns a promise
    .catch(e => {
      return downloadFallbackData(url); // returns a promise
    })
    .then(v => {
      return processDataInWorker(v); // returns a promise
    });
}
```



```
async function getProcessedData(url) {
  let v;
  try {
    v = await downloadData(url);
  } catch(e) {
    v = await downloadFallbackData(url);
  }
  return processDataInWorker(v);
}
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

Kombi aus Promise.all und async / await

```
50  var parallel = async function() {
51    console.log('==PARALLEL with await Promise.all==');
52
53    // Start 2 "jobs" in parallel and wait for both of them to complete
54    await Promise.all([
55        (async()=>console.log(await resolveAfter2Seconds()))(),
56        (async()=>console.log(await resolveAfter1Second()))()
57    ]);
58  }
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

fetch mit async / await

```
async function fetchJson(url) {
 try {
   let request = await fetch(url);
   let text = await request.text();
   return JSON.parse(text);
 }
 catch (error) {
   console.log(`ERROR: ${error.stack}`);
 }
}
```

- Vorteil: Asynchrone Programme fast wie synchrone schreiben können
- Nachteile:
  - ein Promise kann man nicht mehr anhalten
  - Status von Promise nicht abfragbar
  - funktioniert noch nicht so richtig mit Strömen

http://exploringjs.com/es6/ch_generators.html#sec_overview-generators

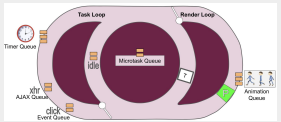
Gliederung

1. AJAX
2. XMLHttpRequest
3. Promise
4. fetch-API
5. async / await
6. Event Loop
7. Events
8. Worker-API

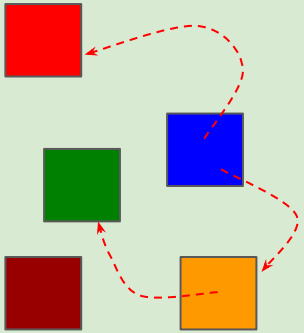
Unterschied Stack, Heap und Event Loop

Browser-Runtime

Event Loop

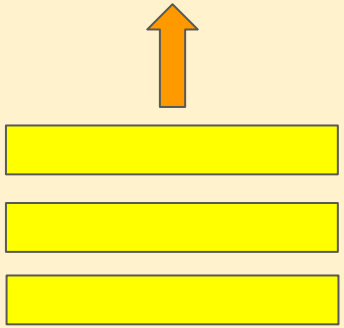


Heap



Closures
Memory Allocation
Garbage Collection

Stack



Execution Context

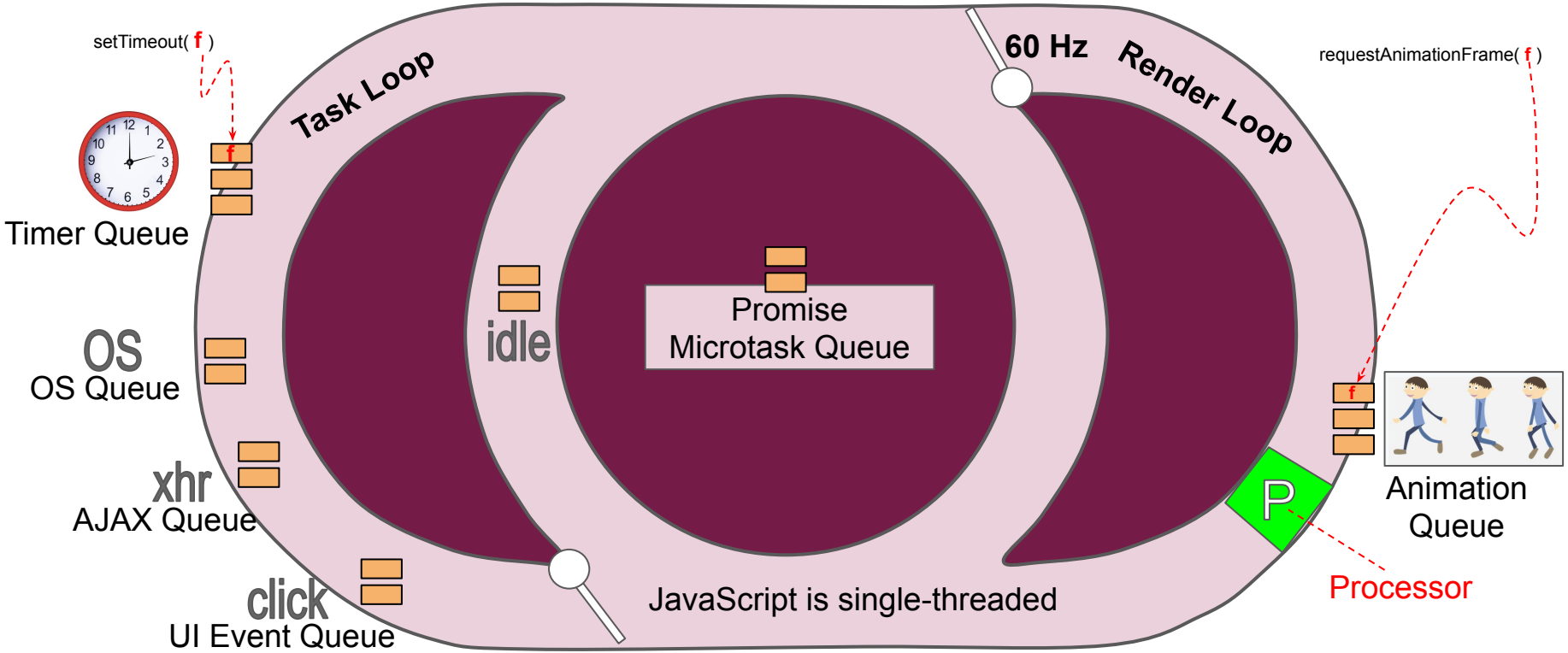
single-threaded:
only one stack
Do one thing at a time.
No concurrency.

Run-to-completion semantics

- Jede Funktion wird bis zu Ende ausgeführt
- völlig ungestört
- keine Nebenläufigkeit während der Ausführung
- Werte werden von außen nicht verändert

Problem: Wie kann man Nebenläufigkeitsartefakte vom Anwendungsprogrammierer fernhalten?

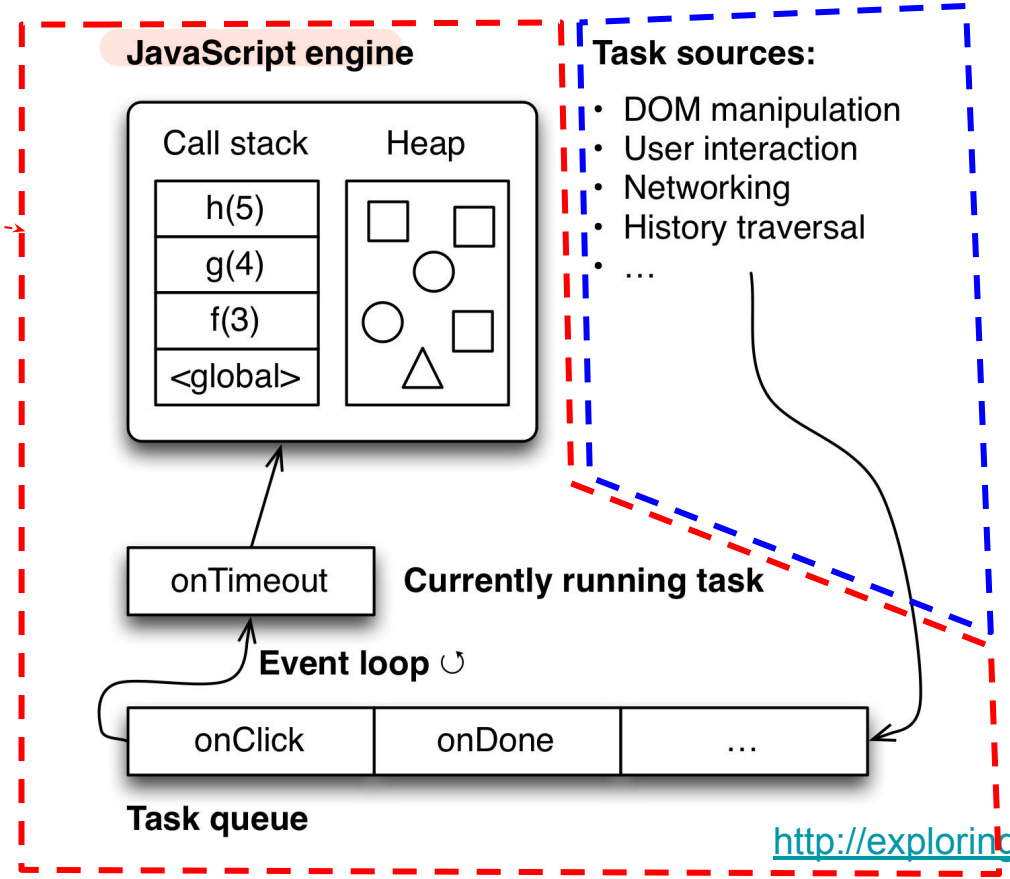
Lösung: Event-Loop in einem Thread in einem Prozess



https://html.spec.whatwg.org/multipage/webappapis.html#event-loop

https://www.youtube.com/watch?v=cCOL7MC4PI0

Single Thread vs. Multithreading vs. Multiprocessing



Der Browser hat für jeden Tab einen eigenen Prozess.

Innerhalb dieses Prozesses gibt es mehrere Threads

für DOM manipulation, User interaction, Networking, …

Jede JavaScript-Funktion ist single-threaded.

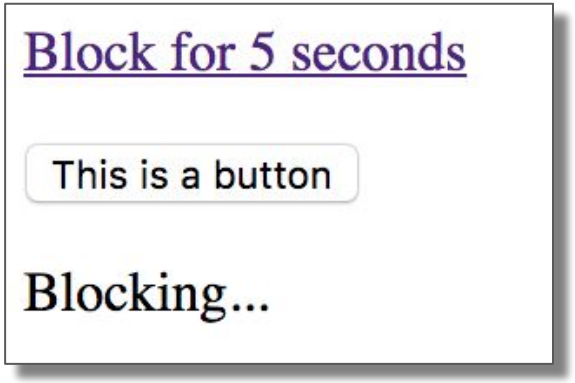
http://exploringjs.com/es6/ch_async.html

Blocking the Event Loop

```
<!doctype html>
<a id="block" href="">Block for 5 seconds</a>
<p>
  <button>This is a button</button>
<div id="statusMessage"></div>
<script>
 document.getElementById('block')
  .addEventListener('click', onClick);

 function onClick(event) {
  event.preventDefault();

  setStatusMessage('Blocking...');

  // Call setTimeout(), so that browser has time to display
  // status message
  setTimeout(function () {
    sleep(5000);
    setStatusMessage('Done');
  }, 0);
 }
```

```
function setStatusMessage(msg) {
  document.getElementById('statusMessage')
      .textContent = msg;
}
function sleep(milliseconds) {
  var start = Date.now();
  while ((Date.now() - start) < milliseconds);
}
</script>
```

Synchronous Timeout



https://kaul.inf.h-brs.de/data/2018/we/demos/09/blocking_event_loop.html

http://exploringjs.com/es6/ch_async.html#sec_receiving-results-asynchronously

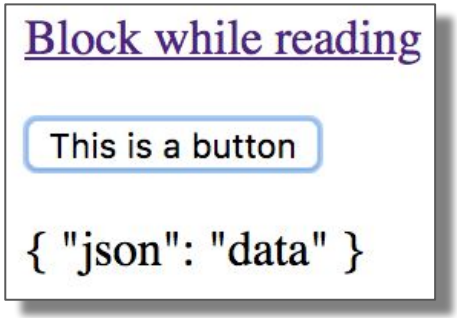
Blocking Event Loop while Reading

```
<!doctype html>
<a id="block" href="">Block while reading</a>
<p>
  <button>This is a button</button>
<div id="statusMessage"></div>
<script>
 document.getElementById('block')
  .addEventListener('click', onClick);

function onClick(event) {
  event.preventDefault();

  setStatusMessage('Blocking...');

  var req = new XMLHttpRequest();
  req.open('GET', './data.json', false );

  req.onload = function () {
    if (req.status == 200) {
      setStatusMessage(req.response);
    } else {
      console.log('ERROR', req.statusText);
    }
  };
```

```
req.onerror = function () {
    console.log('Network Error');
  };

  req.send(); // Add request to task queue
}
function setStatusMessage(msg) {
  document.getElementById('statusMessage')
   .textContent = msg;
}
</script>
```

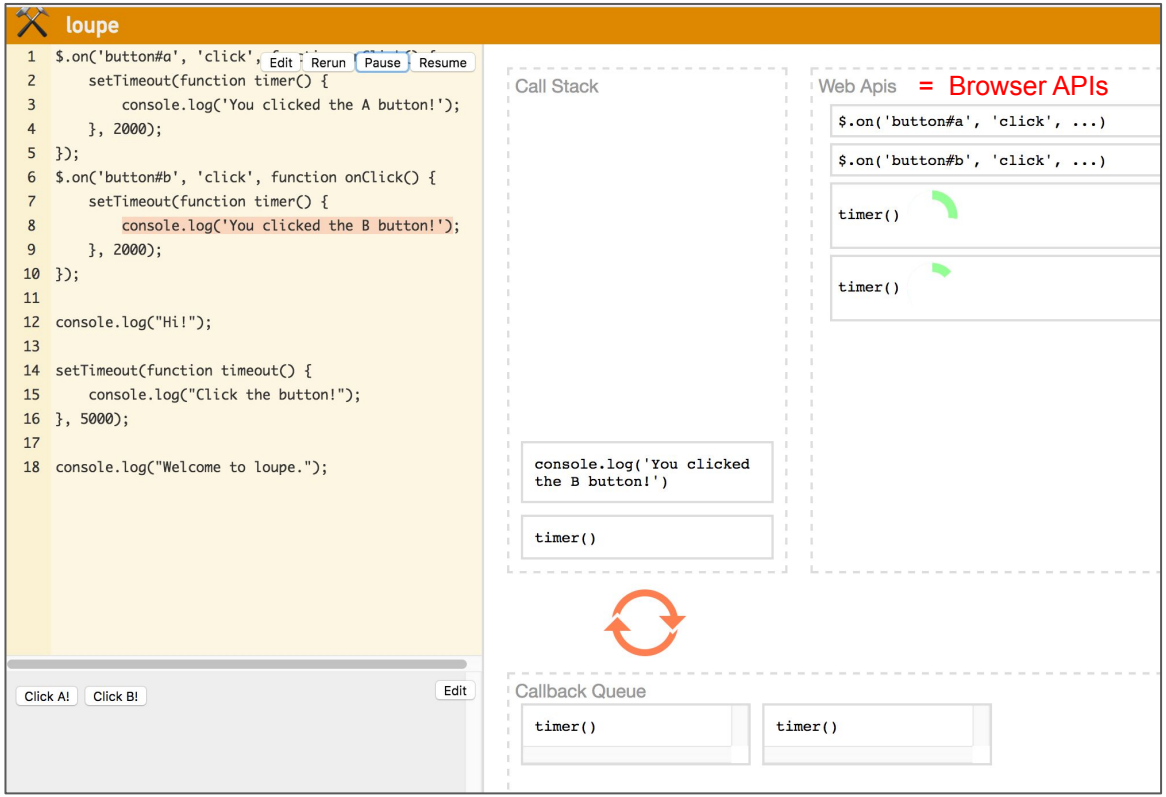
false:
Synchronous
GET-Request

[Deprecation] Synchronous XMLHttpRequest on the main thread is deprecated because of its detrimental effects to the end user's experience. For more help, check https://xhr.spec.whatwg.org/.

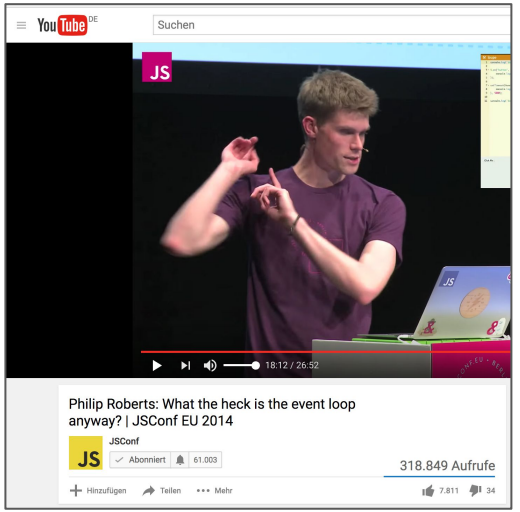


https://kaul.inf.h-brs.de/data/2018/we/demos/09/blocking_while_reading.html

Animation Browser Event Loop



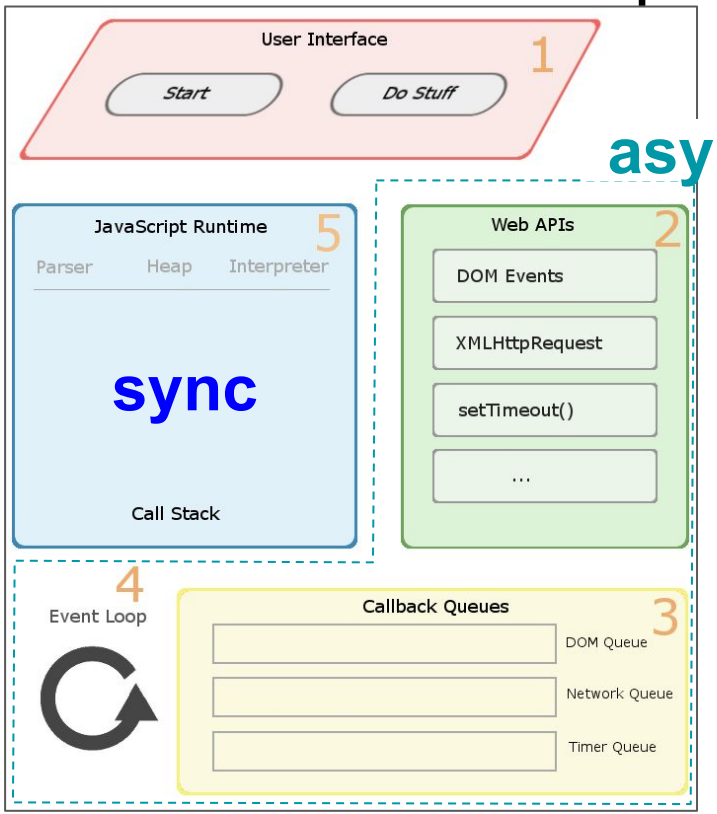
http://latentflip.com/loupe



https://youtu.be/8aGhZQkoFbQ?t=18m12s

https://github.com/latentflip/loupe

Browser Event Loop




```
<button id="doStuff">Do Stuff</button>

<script>
 document.getElementById('doStuff')
   .addEventListener('click', function() {
      console.log('Do Stuff');
    }
   );
</script>
```

Prinzip:
+Anwendungs-
programmierung
einfach, da synchron
+ System-Programmierung
komplex = multi-threaded

Ablauf:

1. Button Click stößt an, dass ...
2. der zugehörige Click Handler unter DOM Events in den Web APIs in die ...
3. Warteschlange "DOM Queue" hinten eingereiht wird.
4. Die Event Loop verwaltet alle Warteschlangen. Sie wartet, dass der Call Stack wieder frei wird und schiebt den nächsten Callback auf den ...
5. Call Stack: Dort wird der Click Handler synchron, single-threaded, single-process, non-blocking (d.h. ohne thread suspense) abgearbeitet.

https://danmartensen.svbtle.com/events-concurrency-and-javascript

Regel: Keine CPU-intensiven Aufgaben in Callbacks

```
<button id="bigLoop">Big Loop</button>
<button id="doStuff">Do Stuff</button>
<script>
document.getElementById('bigLoop')
 .addEventListener('click', function() {
   // big loop
   for (var array = [], i = 0; i < 10000000; i++) {
     array.push(i);
   }
 });

document.getElementById('doStuff')
 .addEventListener('click', function() {
   // message
   console.log('do stuff');
 });
</script>
```

Don’t block the event loop!

Call Stack is single-threaded. ⇒ Während der "big loop" ist der Browser nicht mehr ansprechbar ("eingefroren").

Make sure your code runs at 60 fps (frames-per-second)

Jeder Callback sollte schnell abzuarbeiten sein. Keine CPU-intensiven Aufgaben in Callbacks, Event Handler, Click Handler, ...

⇒ auslagern in "WebWorker"

Anforderung Flickerfreiheit: Geforderte Webseiten-Refresh-Rate von 60 fps

- 60 Frames per second
- = 16 ms per frame
  - 8 ms DOM operation
  - 8 ms Calculation in JavaScript
- Was kann man in 8 ms leisten?
- 8 ms = 8.000.000 ns
  = 8 Mio. Schritte
  bei einer
  1 Ghz-CPU = 1 ns / Takt

Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

https://www.youtube.com/watch?v=Kz_zKXiNGSE

Gliederung

1. AJAX
2. XMLHttpRequest
3. Promise
4. fetch-API
5. async / await
6. Event Loop
7. Events
8. Worker-API

Wdh.: 3 Methoden Event Listener zu definieren

1. via HTML-Attribut

```
<div onclick="changeText(this)">JavaScript wurde noch nicht ausgeführt!</div>
<div id="div2">div2</div>
<div id="div3">div3</div>
<script>
 function changeText( element ){
   element.textContent = "JavaScript wurde jetzt ausgeführt!";
 }

 const div2 = document.getElementById('div2');
 div2.onclick = function( event ){ changeText(this); };

 const div3 = document.getElementById('div3');
 div3.addEventListener('click', function( event ){ changeText(this); } );

 div3.addEventListener( 'click', function( event ){ this.style['background-color'] = '#ffaaaa'; } );

</script>
```

2. via Object Property

2. via Method addEventListener

EventTarget.addEventListener()

The `EventTarget` method **`addEventListener()`** sets up a function that will be called whenever the specified event is delivered to the target. Common targets are `Element`, `Document`, and `Window`, but the target may be any object that supports events (such as `XMLHttpRequest`).

`addEventListener()` works by adding a function or an object that implements `EventListener` to the list of event listeners for the specified event type on the `EventTarget` on which it's called.

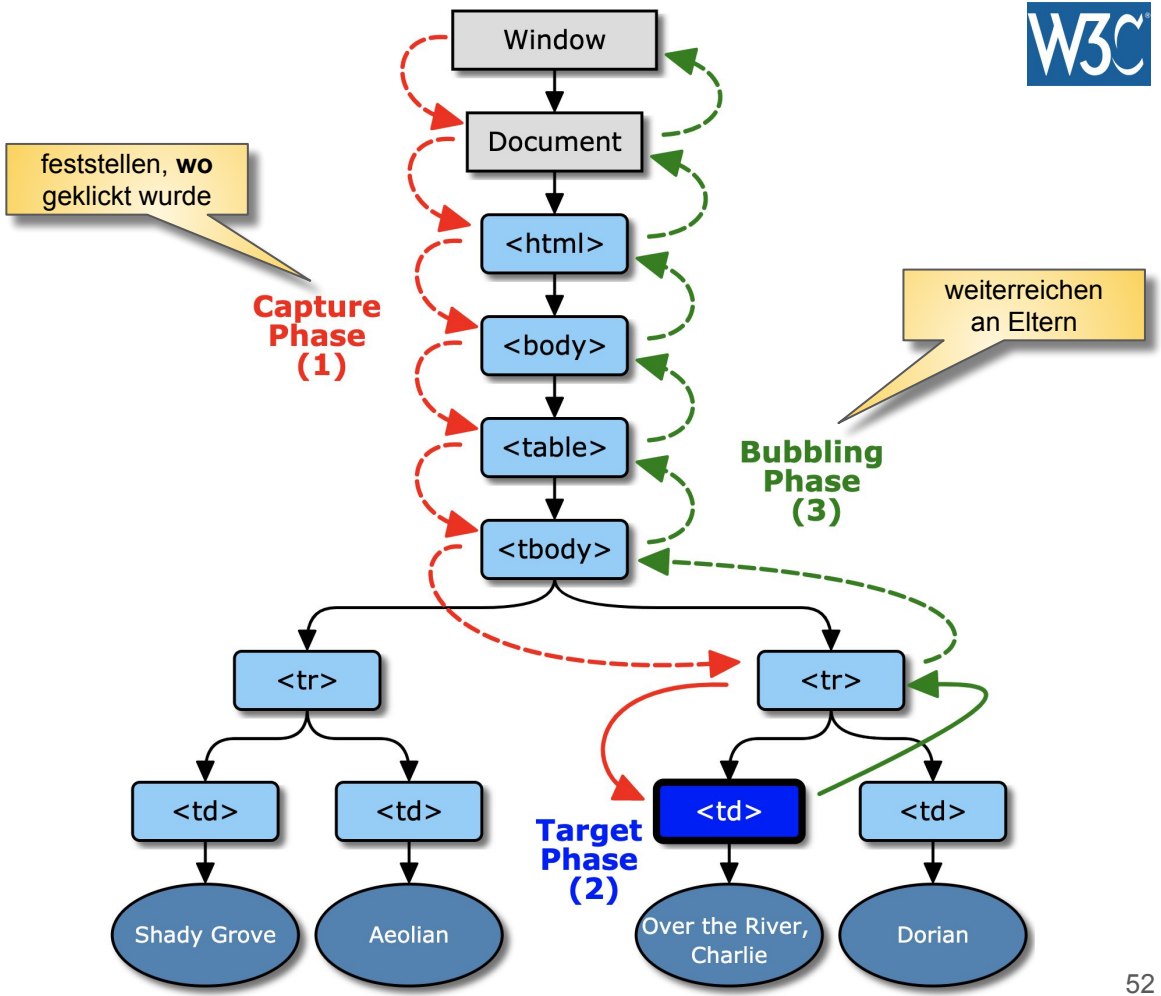
Syntax

```
target.addEventListener(type, listener[, options]);

target.addEventListener(type, listener[, useCapture]);
```

https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener

Event Flow

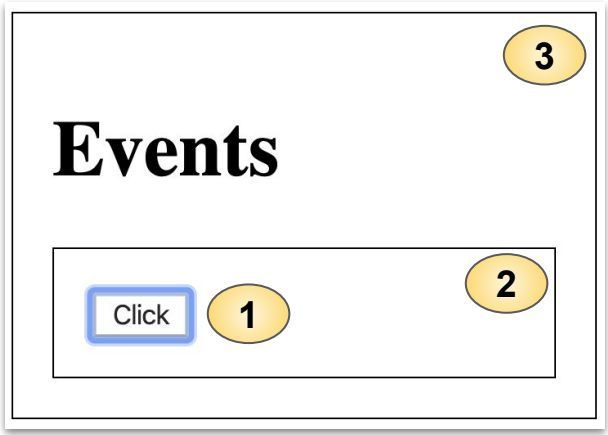
- The **capture phase**: The event object propagates through the target's ancestors from the Window to the target's parent. This phase is also known as the ***capturing phase***.
- The **target phase**: The event object arrives at the event object's event target. This phase is also known as the *at-target phase*. If the event type indicates that the event doesn't bubble, then the event object will halt after completion of this phase.
- The **bubble phase**: The event object propagates through the target's ancestors in reverse order, starting with the target's parent and ending with the Window. This phase is also known as the ***bubbling phase***.

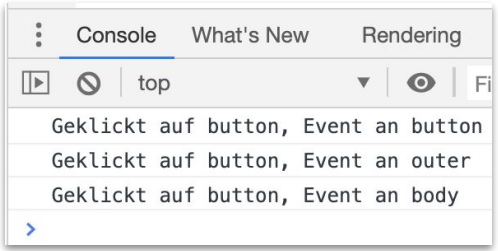


https://www.w3.org/TR/uievents/#event-flow

Bubbling Phase

```
<body id="body">
  <h1>Events</h1>
  <div id="outer">
    <button id="button">Click</button>
  </div>

<script>
  function logger( event ){
      console.log( `Geklickt auf ${event.target.id},
                    Event an ${this.id}` ) }

  ['body','outer','button']
    .map( id => document.getElementById( id ) )
    .forEach( elem => elem.addEventListener('click', logger ));

</script>
</body>
```



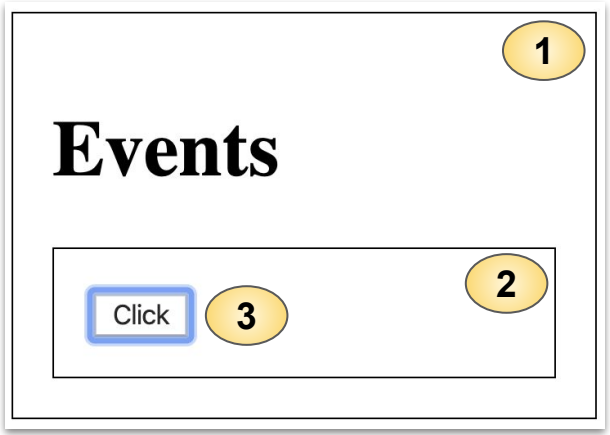


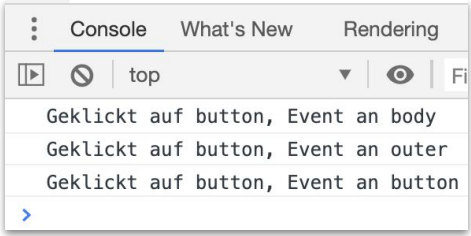
https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener

Capturing Phase

```
<body id="body">
  <h1>Events</h1>
  <div id="outer">
    <button id="button">Click</button>
  </div>

<script>
  function logger( event ){
      console.log( `Geklickt auf ${event.target.id},
                    Event an ${this.id}` ) }

  ['body','outer','button']
    .map( id => document.getElementById( id ) )
    .forEach( elem => elem.addEventListener('click', logger, true ));

</script>
</body>
```

false default

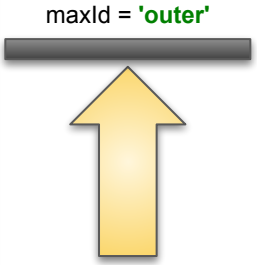


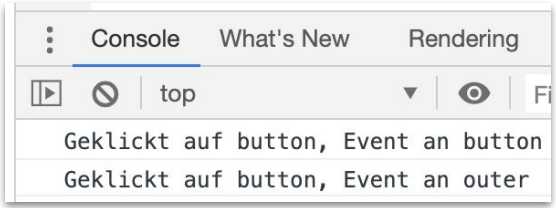


https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener

Event.stopPropagation()

```
<body id="body">
  <h1>Events</h1>
  <div id="outer">
    <button id="button">Click</button>
  </div>
<script>
  const stoppingLogger = maxId => {
    return function ( event ) {
      if (this.id === maxId) event.stopPropagation();
      console.log(`Geklickt auf ${event.target.id}, Event an ${this.id}`)
    }
  };

  ['body','outer','button']
    .map( id => document.getElementById( id ) )
    .forEach( elem => elem.addEventListener('click', stoppingLogger( 'outer' ) ));
</script>
</body>
```





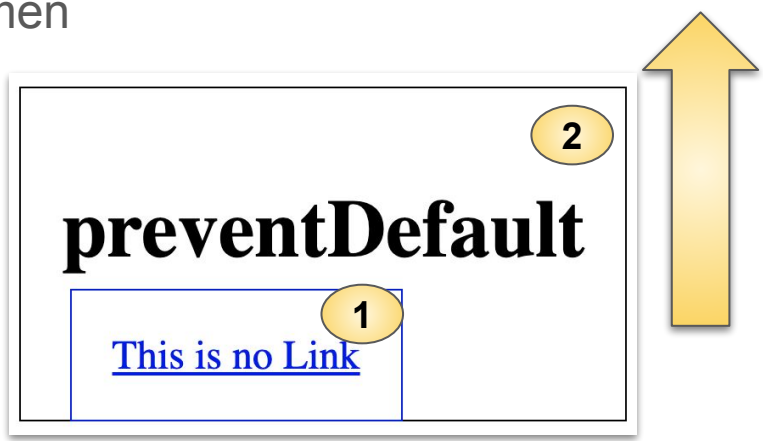
https://developer.mozilla.org/en-US/docs/Web/API/Event/stopPropagation

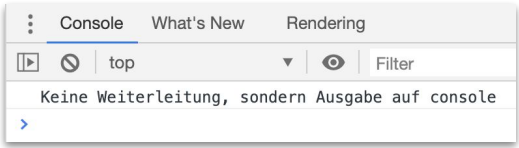
Event.preventDefault()

- das Default-Verhalten des Browsers abklemmen

```
<body>
<h1>preventDefault</h1>
<a href="other.html">This is no Link</a>
<script>
  const a = document.querySelector('a');
  a.addEventListener('click', listener);
  function listener( event ){
    event.preventDefault();
    console.log( 'Keine Weiterleitung, sondern Ausgabe auf console' );
  }
</script>
</body>
```

Event Bubbling findet trotzdem statt.



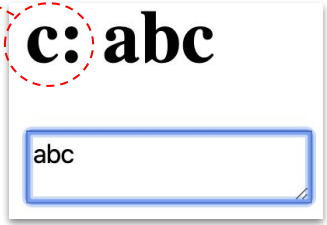


https://developer.mozilla.org/en-US/docs/Web/API/Event/preventDefault

Daten von Events

```
<!doctype html>
<h1 id="title">Title</h1>
<textarea id="start" placeholder="edit..."></textarea>

<script>
 const title = document.getElementById('title');
 const start = document.getElementById('start');

 start.addEventListener("input", function( event ){
   title.textContent = event.data + ": " + this.value;
 });

</script>
```

der zuletzt getippte Buchstabe



Gliederung

1. AJAX
2. XMLHttpRequest
3. Promise
4. fetch-API
5. async / await
6. Event Loop
7. Events
8. Worker-API

WebWorker

- JavaScript ist Single-Threaded.
- Web Worker werden in einem separaten Thread ausgeführt.

index.html

```
<!DOCTYPE html>
<body>
  <button>Start</button>
  <script>
    const worker = new Worker('./doWork.js');
    const button = document.querySelector('button');
    button.addEventListener('click', function(e){
      worker.postMessage('Hello World'); // start worker.
    });

    worker.addEventListener('message', function(e){
      document.body.innerHTML += 'Worker said: ' + e.data;
    });
  </script>
</body>
```

./doWork.js

```
self.addEventListener('message', function(e) {
 for (var array = [], i = 0; i < 100_000; i++) {
   array.push(i);  // do some CPU intensive work
 }
 self.postMessage(e.data);
});
```

https://www.w3.org/TR/workers/
https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API
https://www.html5rocks.com/de/tutorials/workers/basics/

Zusammenfassung

1. AJAX
2. XMLHttpRequest
3. Promise
4. fetch-API
5. async / await
6. Event Loop
7. Events
8. Worker-API