

AJAX IN PLAIN JAVASCRIPT

LERNZIELE

- Was ist Ajax?
- Was sind Callbacks?
- Wie funktioniert Synchroner und Asynchroner Code in Javascript?
- Wie rufe ich in Javascript Daten von einem Server ab?
- Wie machen uns Promises das Leben leichter?
- Was ist async/await?

Speaker notes

- Lernziel für heute ist das Ajax Prinzip zu verstehen und in einer Javascript Application umsetzen zu können.
- Callbacks sind die Grundlage für asynchrone Programmierung in Javascript, daher müssen wir uns dies als erstes ansehen.
- Synchronen Code sollte jeder bereits geschrieben haben, evtl. habt ihr auch in Java schon mit Threads asynchron gearbeitet.
- In Javascript ist asynchron nicht gleich asynchron und funktioniert auch grundlegend anders als in Java, daher wollen wir uns dies noch mal genauer ansehen.
- Wenn wir dies verstanden haben, können wir mal einen ersten Aufruf Richtung Backend starten.
- Promises sind Konstrukte, die uns die Arbeit mit asynchronität erleichtern können, um dies zu nutzen müssen wir verstehen wie.
- Async/await setzt dem ganzen dann die Krone auf.

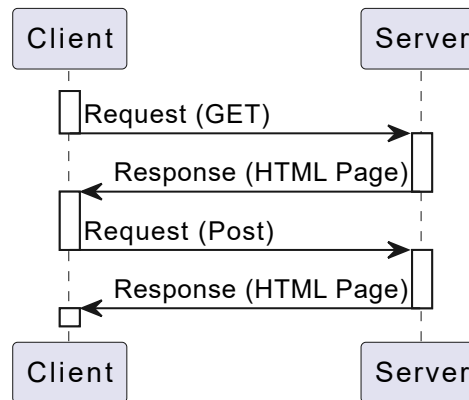
AJAX

(Asynchronous JavaScript and XML)

Speaker notes

- Ganz wichtig: Ajax hat erstmal nichts direkt mit XML zu tun.
- Der Name Ajax ist in Bezug auf das XML ziemlich fehlleitend.
- Wir klären jetzt genau was dahinter steckt.

MULTI PAGE APPLICATION (ERINNERUNG)



Speaker notes

- Jedes mal wenn der Nutzer andere Daten sehen möchte, auf eine andere Seite navigieren möchte oder ein Formular abschicken will, wird ein Request ans Backend geschickt und er erhält als Antwort eine neue HTML Seite geschickt.
- Während der Server die Antwort vorbereitet, kann der Nutzer nicht mit der Seite interagieren.
- Erst wenn der Server diese neue Seite zurückschickt kann der Browser das Ergebnis anzeigen.

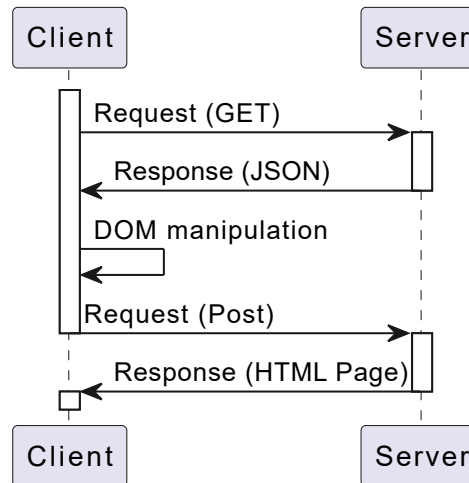
WAS IST AJAX?

- es ist keine Programmiersprache!
- es handelt sich um ein Programmierprinzip
- Daten anzeigen, ohne eine neue HTML Seite auszuliefern

Speaker notes

- Die Idee ist dem Nutzer andere Daten anzuzeigen, ohne eine Komplette neue HTML Seite zu laden.

AJAX



Speaker notes

- Mit dem Ajax Prinzip könnte das vorherige Schaubild etwa so aussehen.
- Während die HTML Seite angezeigt wird, werden mit JavaScript im Hintergrund Daten vom Server abgefragt.
- Kommen die Daten zurück, muss das HTML manipuliert werden, damit die Daten angezeigt werden.
- Wenn sowieso die ganze Seite ausgetauscht werden muss, können wir natürlich weiterhin einfach einen Seitenaufruf ausführen.
- Wir schauen uns jetzt Stück für Stück an, wie wir eine Application mit dem Ajax Prinzip bauen können.

CALLBACKS

WAS SIND CALLBACKS?

“In computer programming, a callback is a piece of executable code that is passed as an argument to other code, which is expected to call back (execute) the argument at some convenient time. The invocation may be immediate as in a synchronous callback or it might happen at later time, as in an asynchronous callback.”

Wikipedia - Callback (computer programming)

Speaker notes

- Hier ist auch von asynchronen Callbacks die Rede, darauf gehen wir später genauer ein.
- Um es einfacher zu machen, schauen wir uns jetzt erstmal synchrone Callbacks auf.
- Callback Zitat: [https://en.wikipedia.org/wiki/Callback_\(computer_programming\)](https://en.wikipedia.org/wiki/Callback_(computer_programming))

BEISPIEL CALLBACK:

```
1 function someFunction(callback) {
2     console.log('do something');
3
4     callback()
5 }
6
7 function someCallbackFunction() {
8     console.log('callback called');
9 }
10
11 function onClick() {
12     someFunction(someCallbackFunction)
13 }
```

Speaker notes

- Hier sehen wir, wie Funktionen in Javascript als Parameter in eine andere Funktion gegeben werden können.
- Dies nennt man dann einen Callback.
- Bitte an dieser Stelle den Code im Repository

PRAXIS

Code aus dem Repo auschecken:

`gitlab.com/dhbw_webengineering_2/plain_javascript`

Branch: "step_1"

Speaker notes

- Wir schauen uns verschiedene Callbacks in richtigen Javascript Code an, um es besser zu vertiefen.
- Alle Beispiele habe ich euch auch hier im Foliensatz noch mal aufgeführt, damit ihr sie für später noch einmal nachsehen könnt.

BEISPIEL CALLBACK ANONYM:

```
1 function someFunction(callback) {  
2     console.log('do something');  
3  
4     callback()  
5 }  
6  
7  
8 function onClick() {  
9     someFunction(() => console.log('callback called'));  
10 }
```

BEISPIEL CALLBACK MIT PARAMETER:

```
1 function someFunction(callback) {  
2     console.log('do something');  
3  
4     callback('callback called');  
5 }  
6  
7 function someCallbackFunction(string) {  
8     console.log(string);  
9 }  
10  
11 function onClick() {  
12     someFunction(someCallbackFunction);  
13 }
```

BEISPIEL CALLBACK MIT PARAMETER ANONYM:

```
1 function someFunction(callback) {  
2     console.log('do something');  
3  
4     callback('callback called');  
5 }  
6  
7 function onClick() {  
8     someFunction((string) => console.log(string));  
9 }
```


PROMISES

PROMISES

- erleichtern uns das Arbeiten mit asynchronem Code
 - sie machen aus einem Callback wieder einen "regulären" Rückgabewert
 - dieser Rückgabewert ist Zeitversetzt
 - es wird ein "Versprechen" gegeben, dass ein Wert zurückgegeben wird

PROMISES VERWENDUNG:

```
1 function callBackend() {  
2     return new Promise();  
3 }
```

```
1 const promise = callBackend();  
2 promise  
3     .then((result) => console.log(result))  
4     .catch((error) => console.error(error));
```

Speaker notes

- Promises werden wie normale Datentypen (string, number, date, etc.) aus einer Funktion zurückgegeben.
- damit sieht der Programmfluss schon mal "normaler" aus.
- wenn ich das Ergebnis des Promises haben möchte
 - rufe ich .then auf, um einen Callback einzufügen, der das Ergebnis verarbeitet.
 - rufe ich .catch auf, um im Fehlerfall ein Fehlerhandling hinzuzufügen.
- Jetzt geben wir wieder Callbacks in das Promise, haben wir damit etwas gewonnen?
- Dazu später mehr...

PROMISES BEISPIEL:

- Wichtig: an dem XMLHttpRequest ändert sich erstmal nichts
- der Callback muss also in ein Promise umgewandelt werden

```
1 function get() {
2     return new Promise((resolve, reject) => {
3         const xhr = new XMLHttpRequest();
4         xhr.onload = () => {
5             if (xhr.status >= 200 && xhr.status < 300) {
6                 resolve(JSON.parse(xhr.response));
7             } else {
8                 reject(xhr.statusText);
9             }
10        }
11    });
12 }
```

Speaker notes

- Wenn man ein neues Promise erstellt, bekommt man in einem Callback eine resolve und eine reject Funktion.
 - resolve nimmt im positiven Fall das Ergebnis zurück.
 - mit reject wird ein Fehler zurückgegeben.
- In diesem Fall gebe ich das Ergebnis vom Backend zurück, wenn es geklappt hat oder den Statustext, wenn ein Fehler aufgetreten ist.
- Wahrscheinlich sieht dies für einige erstmal so aus, als würden Promises einiges komplizierter machen.
- Das ist auch so, Promises bringen aber auch Vorteile mit sich, sonst würde sie ja keiner verwenden.

PROMISES BEISPIEL:

- Wichtig: an dem XMLHttpRequest ändert sich erstmal nichts
- der Callback muss also in ein Promise umgewandelt werden

```
1 function get() {  
2     return new Promise((resolve, reject) => {  
3         const xhr = new XMLHttpRequest();  
4         xhr.onload = () => {  
5             if (xhr.status >= 200 && xhr.status < 300) {  
6                 resolve(JSON.parse(xhr.response));  
7             } else {  
8                 reject(xhr.statusText);  
9             }  
10        }  
11    });  
12 }
```

PROMISES BEISPIEL:

- Wichtig: an dem XMLHttpRequest ändert sich erstmal nichts
- der Callback muss also in ein Promise umgewandelt werden

```
1 function get() {
2     return new Promise((resolve, reject) => {
3         const xhr = new XMLHttpRequest();
4         xhr.onload = () => {
5             if (xhr.status >= 200 && xhr.status < 300) {
6                 resolve(JSON.parse(xhr.response));
7             } else {
8                 reject(xhr.statusText);
9             }
10        }
11    });
12 }
```

CALLBACK HELL

```
1 a((resultFromA) => {
2     b(resultFromA, (resultFromB) => {
3         c(resultFromB, (resultFromC) => {
4             d(resultFromC, (resultFromD) => {
5                 console.log(resultFromD);
6             });
7         });
8     });
9 });
```

```
1 a().then((result) => {
2     return b(result);
3 }).then((result) => {
4     return c(result);
5 }).then((result) => {
6     return d(result);
7 }).then((result) => {
8     console.log(result);
9 });
```


Speaker notes

- Aufeinander aufbauende Operationen können mit Callbacks schnell tiefe Einrückungen erzeugen.
- Promises können einfach aneinandergehängt werden. Damit wird der Code einfacher zu lesen.
- Kein Problem mit Namespaces beim Promise chaining.
 - Im oberen Beispiel sollten wir für jedes "result" einen anderen Namen nutzen, da das dies sonst für Verwirrung sorgen könnte.
#
 - Wird der gleich Variablenname verwendet, "shadowen" wir die Variable einen "Scope" darüber.

PROMISES METHODEN

- Promise.all()
- Promise.allSettled()
- Promise.any()
- ...

```
1 Promise.all([get(url), get(url)]).then(([joke1, joke2]) => {  
2     document.getElementById('joke1').innerHTML = joke1.joke;  
3     document.getElementById('joke2').innerHTML = joke2.joke;  
4 }).catch((error) => console.log(error));
```

Speaker notes

- Mit Promises können wir recht einfach unser Problem von vorhin lösen. Mehrere Backendcalls können wir mit `Promise.all()` parallel abschicken und warten bis alle geantwortet haben.
- `Promise.all()` resolved nur, wenn alle Promises resolved wurden.
- `Promise.allSettled()` resolved auch, wenn einzelne rejected wurden.
- `Promise.any()` resolved, wenn ein Promise resolved wurde.
- Weitere Infos findet ihr hier: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
- Ist aber nicht tiefergehend Prüfungsrelevant.

VORTEILE VON PROMISES

- Callback Hell wird vermieden
- Fehler werden expliziter behandelt
- Promises Methoden helfen bei asynchronem Code

PRAXIS

- Branch: "step_5" - ein Beispiel für Promises
- Vermeidet die kleine "callback hell" in Branch "step_5"
- Aufgabe von vorhin (Paralleler Request HTML editieren erst nach dem resollen beider Requests) jetzt mit Promises
- Branch: "step_6" - ein Beispiel für Promise.all() (Lösung)

ASYNC/AWAIT

SYNTAKTISCHER ZUCKER

- `asnc/await` macht Promises noch schöner
- mit `async` können wir Funktionen markieren, die asynchronen Code enthalten
- mit `await` warten wir auf ein Ergebnis eines Promises
- mit Codebeispielen versteht man es einfacher

CODEBEISPIEL

```
1 async function showDadJoke() {  
2     const url = 'https://icanhazdadjoke.com/';  
3  
4     const joke1 = await get(url);  
5     const joke2 = await get(url);  
6  
7     document.getElementById('joke1').innerHTML = joke1.joke;  
8     document.getElementById('joke2').innerHTML = joke2.joke;  
9 }
```


Speaker notes

- Mit `async` markiere ich, dass die Funktion `showDadJoke()` asynchronen Code enthält.
- Damit gebe ich dem Aufrufer die Möglichkeit darauf zu warten oder sie asynchron laufen zu lassen.
- Mit `await` warte ich auf das `resolves` der `Promises`.
- Die Variablen `joke1` und `joke2` enthalten daher kein `Promise`, sondern das Ergebnis des `Promises`.
- Unten kann ich dann einfach auf das Ergebnis zugreifen.
- Jetzt sieht unser asynchroner Code fast wieder "normal" aus.

CODEBEISPIEL

```
1 async function showDadJoke() {  
2     const url = 'https://icanhazdadjoke.com/';  
3  
4     const joke1 = await get(url);  
5     const joke2 = await get(url);  
6  
7     document.getElementById('joke1').innerHTML = joke1.joke;  
8     document.getElementById('joke2').innerHTML = joke2.joke;  
9 }
```

CODEBEISPIEL

```
1 async function showDadJoke() {  
2     const url = 'https://icanhazdadjoke.com/';  
3  
4     const joke1 = await get(url);  
5     const joke2 = await get(url);  
6  
7     document.getElementById('joke1').innerHTML = joke1.joke;  
8     document.getElementById('joke2').innerHTML = joke2.joke;  
9 }
```

CODEBEISPIEL

```
1 async function showDadJoke() {  
2     const url = 'https://icanhazdadjoke.com/';  
3  
4     const joke1 = await get(url);  
5     const joke2 = await get(url);  
6  
7     document.getElementById('joke1').innerHTML = joke1.joke;  
8     document.getElementById('joke2').innerHTML = joke2.joke;  
9 }
```

PRAXIS

- Branch: "step_7" - enthält ein Beispiel für async/await
- Werden die Calls jetzt nacheinander abgeschickt?
- Wie bekommen wir sie wieder parallel?
- Lösung gibt es auch Branch: "step_8"
- Nehmt euch auch hier gerne die Zeit und spielt etwas damit herum.