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.

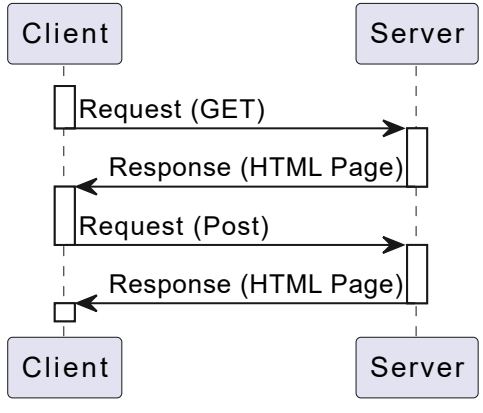
AJAX

(Asynchronus 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.

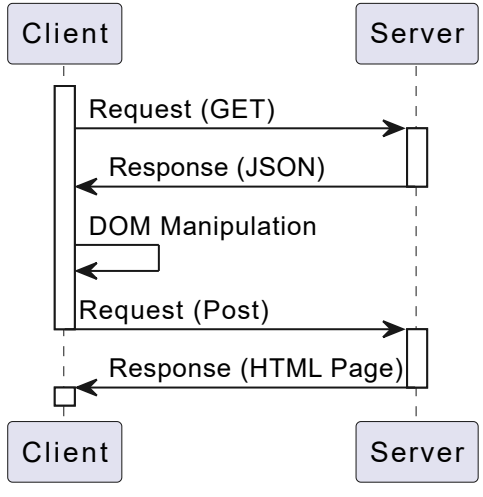
WAS IST AJAX?

- 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 Komplett neue HTML Seite zu laden.

AJAX


Client
Server
Request (GET)
Response (JSON)
DOM Manipulation
Request (Post)
Response (HTML Page)
Client
Server

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 ausgetausch 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 asynchonen 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)

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 onButtonClick() {
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 onButtonClick() {
 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 onButtonClick() {
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 onButtonClick() {
8     someFunction((string) => console.log(string));
9 }
```

SYNCHRON UND ASYNCHRON

SYNCHRON

- Programmabschnitte werden der Reihe nach ausgeführt
- im Code ist bereits festgehalten wie diese Reihenfolge aussieht
- zur Laufzeit kann sich diese nicht ändern

ASYNCHRON

- Programmschritte werden nicht Synchron ausgeführt
- es muss nicht auf Programmschritte gewartet werden
- Programmschritte können gleichzeitig ausgeführt werden (nicht in JS)

Speaker notes

- Wenn wir Programmschritte haben, die länger dauern oder auf einen Request vom Backend warten, können wir in der zwischenzeit bereits anderen Code ausführen.
- In Java können wir über verschiedene Threads rechenintensive Programmschritte, die unabhängig voneinander sind, parallel ausführen.
- In Javscript ist dies nicht möglich. Javascript hat nur einen einzigen Thread zu verfügung. Damit ist echte parallelität nicht verfügbar.
- Wir können uns asynchonität trotzdem zur nutze machen, da wir bereits Code ausführen können, wenn wir noch auf einen Backend request warten. Schließlich ist während dem Request keine Berechnung notwendig.
- Um das besser zu verstehen, schauen wir uns dies wieder in der Praxis an.

BEISPIEL:

```
 1 function synchron() {
 2     console.log("step 1");
 3     console.log("step 2");
 4     console.log("step 3");
 5 }
 6
 7 function asynchron() {
 8     console.log("step 1");
 9     setTimeout(() => {
10         console.log("step 2");
11     }, 1000);
12     console.log("step 3");
13 }
```

Speaker notes

- Im synchronen Beispiel wird nacheinander "step 1", "step 2" und "step 3" auf der Konsole geschrieben.
- Im asynchronen Beispiel ist dies nicht der Fall. Checkt mal im Plain Javascript Repository den Branch "step_2" aus und probiert es selbst.
- Die Funktion "setTimeout" simuliert eine längere Prozedur.
- "setTimeout" nimmt einen Callback entgegen, der nach einer gewissen Zeit aufgerufen wird.
- Die Zahl, die wir übergeben, definiert die Millisekunden, die mindestens gewartet werden, bis der Callback aufgerufen wird.
- Warum mindestens?
- Wie wir gelernt haben, haben wir in Javascript kein Multithreading. Das heißt alles wird auf einem Thread ausgeführt. Es kann daher passieren, dass nach 1000 Millisekunden grade etwas anderes ausgeführt wird und unser Callback warten muss.
- Am besten wir spielen noch etwas mit "setTimeout" herum, um die asynchronität besser zu verstehen.

JAVASCRIPT EVENT LOOP

Jake Archibald: In The Loop - setTimeout, micro tasks, requestAnimationFrame, requestIdleCallback, ...

Speaker notes

- Da es in Javascript nur einen Thread gibt, müssen die verschiedenen Tasks nacheinander abgearbeitet werden.
- Damit es hier nicht zu Lags für den Nutzer kommt, gibt es die Event Loop.
- Tasks werden in diese Event Loop eingestellt und nacheinander abgearbeitet.
- Zwischen jedem Task wird einmal durch die Event Loop gekreist. Damit könnte ein Rerender ausgelöst werden.
- Damit wird sichergestellt, dass die UI zwischenzeitlich neu gezeichnet wird und der Nutzer keine Lags hat.
- Normale asynchrone Tasks werden auf der so genannten Macroqueue eingestellt. Pro Event Loop cycle wird ein Macrotask ausgeführt.
- Damit wird sichergestellt, dass zwischendrin immer ein Rerender passieren kann.
- Für wichtigere Tasks gibt es noch mal die Microqueue. Tasks der Microqueue werden zum nächstmöglichen Zeitpunkt ausgeführt.
- Sobald kein Task mehr ausgeführt wird und auch kein Rerender stattfindet werden solange Tasks von der Microqueue ausgeführt bis nichts mehr in der Microqueue ist.
- Callbacks eines Promises landen auf der Microqueue. Zu Promises später mehr.
- Schaut euch gerne noch mal das Video ausführlich an, dann versteht ihr im Detail wie die Event Loop funktioniert.

PRAXIS

Branch: "step_2"

DATEN VOM BACKEND LADEN

XML HTTP REQUEST

```
 1 function showDadJoke() {
 2     const xhr = new XMLHttpRequest();
 3     xhr.onload = () => {
 4         const joke = JSON.parse(xhr.response);
 5         document.getElementById('joke').innerHTML = joke.jo
 6     }
 7     xhr.open('GET', 'https://icanhazdadjoke.com/', true);
 8     xhr.setRequestHeader('Accept', 'application/json');
 9     xhr.send();
10 }
```

Speaker notes

- Daten können wir in Javascript mit einem XMLHttpRequest aus dem Backend laden.
- Der "onload" Parameter ist ein Callback.
  - Wie wir zuvor gelernt haben, wird die Funktion die wir hier übergeben zu späterem Zeitpunkt aufgerufen.
  - Genauer gesagt wird die Funktion aufgerufen, wenn wir eine Antwort vom Backend erhalten haben.
- Der "open" Funktion können wir die HTTP Methode und die URL übergeben, damit das XMLHttpRequest Objekt weis, was es aufrufen soll.
  - Backendrequests können zum teil etwas länger dauern. Möglicherweise wollen wir nicht so lange auf eine Antwort vom Backend warten.
  - Der letzte Parameter der "open" Funktion gibt an, ob der Backendrequest asynchron oder synchron laufen soll.
  - Wie wir auch zuvor gelernt haben, brauchen wir bei asynchronem Code nicht zu warten und es kann bereits weiterer Code ausgeführt werden.
  - Testet dies gerne mal in der nachfolgenden Übung aus.
- In unserem Fall müssen wir der API angeben, welchen Datentyp wir zurückerwarten.
- Dies machen wir mit einem Request Header.
- Die "send" Funktion startet abschließend den Backendrequest.

XML HTTP REQUEST

```
function showDadJoke() {
    const xhr = new XMLHttpRequest();
    xhr.onload = () => {
        const joke = JSON.parse(xhr.response);
        document.getElementById('joke').innerHTML = joke.jo
    }
    xhr.open('GET', 'https://icanhazdadjoke.com/', true);
    xhr.setRequestHeader('Accept', 'application/json');
    xhr.send();
}
```

XML HTTP REQUEST

```
1  function showDadJoke() {
2      const xhr = new XMLHttpRequest();
3      xhr.onload = () => {
4          const joke = JSON.parse(xhr.response);
5          document.getElementById('joke').innerHTML = joke.jo
6      }
7      xhr.open('GET', 'https://icanhazdadjoke.com/', true);
8      xhr.setRequestHeader('Accept', 'application/json');
9      xhr.send();
10 }
```

XML HTTP REQUEST

```
function showDadJoke() {
    const xhr = new XMLHttpRequest();
    xhr.onload = () => {
        const joke = JSON.parse(xhr.response);
        document.getElementById('joke').innerHTML = joke.jo
    }
    xhr.open('GET', 'https://icanhazdadjoke.com/', true);
    xhr.setRequestHeader('Accept', 'application/json');
    xhr.send();
}
```

XML HTTP REQUEST

```
function showDadJoke() {
    const xhr = new XMLHttpRequest();
    xhr.onload = () => {
        const joke = JSON.parse(xhr.response);
        document.getElementById('joke').innerHTML = joke.jo
    }
    xhr.open('GET', 'https://icanhazdadjoke.com/', true);
    xhr.setRequestHeader('Accept', 'application/json');
    xhr.send();
}
```

XML HTTP REQUEST

```
function showDadJoke() {
    const xhr = new XMLHttpRequest();
    xhr.onload = () => {
        const joke = JSON.parse(xhr.response);
        document.getElementById('joke').innerHTML = joke.jo
    }
    xhr.open('GET', 'https://icanhazdadjoke.com/', true);
    xhr.setRequestHeader('Accept', 'application/json');
    xhr.send();
}
```

PRAXIS

Branch: "step_3"

Speaker notes

- Testet gerne mal mit dem async Parameter der "open" Funktion.
- Wie verhält sich darauffolgender Code?

PRAXIS: MEHRERE BACKENDREQUESTS

- Schickt mehrere Backendrequests nacheinander ab
- Falls ihr nicht weiter wisst: Branch "step_4"
- Wie kann ich die Requests parallel abschicken, um Zeit zu sparen?

NETZWERKKONSOLE

- Schaut in die Netzwerkkonsole eures Browsers, wenn ihr mit den verschiedenen Calls testet
- Hier bekommt ihr angezeigt wie lange der Request dauert
- Ihr könnt außerdem sehen, ob sie nacheinander oder parallel abgeschickt werden

PRAXIS: ADVANCED

Wie muss der Code aussehen, wenn wir sie gleichzeitig senden wollen, wir aber den HTML Code erst anpassen wollen, wenn beide Witze zurückgekommen sind?

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

```
function get() {
    return new Promise((resolve, reject) => {
        const xhr = new XMLHttpRequest();
        xhr.onload = () => {
            if (xhr.status >= 200 && xhr.status < 300) {
                resolve(JSON.parse(xhr.response));
            } else {
                reject(xhr.statusText);
            }
        }
    });
}
```

PROMISES BEISPIEL:

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

```
function get() {
    return new Promise((resolve, reject) => {
        const xhr = new XMLHttpRequest();
        xhr.onload = () => {
            if (xhr.status >= 200 && xhr.status < 300) {
                resolve(JSON.parse(xhr.response));
            } else {
                reject(xhr.statusText);
            }
        }
    });
}
```

CALLBACK HELL

```
a((resultFromA) => {
    b(resultFromA, (resultFromB) => {
        c(resultFromB, (resultFromC) => {
            d(resultFromC, (resultFromD) => {
                console.log(resultFromD);
            });
        });
    });
});
```

```
a().then((result) => {
    return b(result);
}).then((result) => {
    return c(result);
}).then((result) => {
    return d(result);
}).then((result) => {
    console.log(result);
});
```

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 mir 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 resolven beider Requests) jetzt mit Promises
- Branch: "step_6" - ein Beispiel für Promise.all() (Lösung)

PRAXIS: PROMISES UND DIE EVENT LOOP

- Promises sind Microtasks
- sie landen daher auf der Microqueue
- Tasks in der Microqueue werden nächstmöglich bearbeitet
- Testet mal setTimeout() vs Promise.resolve().then()

```
1 console.log('start');
2
3 setTimeout(() => console.log('setTimeout'));
4
5 Promise.resolve().then(() => console.log('Promise'));
6
7 console.log('end');
```

Speaker notes

- Sobald der letzte Task von der Javascript Event Loop bearbeitet wurde und ein Microtask in der Microqueue ist, wird direkt der Microtask ausgeführt.
- Dies passiert solange, bis kein Microtask mehr in der Queue ist.
- Die Microqueue hat eine höhere Priorität als normale Tasks und ein Rerender der UI.
- Führt den aufgeführten Code mal in eurem Browser aus. Was hättet ihr erwartet?

ASYNC/AWAIT

SYNTAKTISCHER ZUCKER

- asnyc/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

```
async function showDadJoke() {
    const url = 'https://icanhazdadjoke.com/';

    const joke1 = await get(url);
    const joke2 = await get(url);

    document.getElementById('joke1').innerHTML = joke1.joke;
    document.getElementById('joke2').innerHTML = joke2.joke;
}
```

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 resolven 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.
- Hier wird allerdings ein typischer Fehler gemacht.
- Die Backendcalls werden nacheinander abgeschickt.
- Dies müssen sie nur, wenn sie voneinander Abhängig sind, ansonsten sollten wir sie parallel abschicken.
- Ihr könnt später probieren, wie es besser klappt.

CODEBEISPIEL

```
async function showDadJoke() {
    const url = 'https://icanhazdadjoke.com/';

    const joke1 = await get(url);
    const joke2 = await get(url);

    document.getElementById('joke1').innerHTML = joke1.joke;
    document.getElementById('joke2').innerHTML = joke2.joke;
}
```

CODEBEISPIEL

```
async function showDadJoke() {
    const url = 'https://icanhazdadjoke.com/';

    const joke1 = await get(url);
    const joke2 = await get(url);

    document.getElementById('joke1').innerHTML = joke1.joke;
    document.getElementById('joke2').innerHTML = joke2.joke;
}
```

CODEBEISPIEL

```
async function showDadJoke() {
    const url = 'https://icanhazdadjoke.com/';

    const joke1 = await get(url);
    const joke2 = await get(url);

    document.getElementById('joke1').innerHTML = joke1.joke;
    document.getElementById('joke2').innerHTML = joke2.joke;
}
```

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.

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

- Ajax: Daten asynchron von einem Server laden.
- Callback: Wie ein Rückruf bei einem Telefon.
- Synchronität und Asynchronität (auf einem Thread) -> Überbrückung von Wartezeit.
- Backend aufrufen mit XMLHttpRequests.
- Promises, eine schöne Syntax für Asynchronität.
- Async/await syntaktischer Zucker für Promises.