Hochschule
Albstadt-Sigmaringen

Albstadt-Sigmaringen University

Web2.0 AJAX

Dipl. Ing. Sven Eppler (FH)
sodge IT GmbH

Wie das Web interaktiver wurde

Traditionelle Web Anwendungen

- **Jeder Event triggert einen kompletten Page-Load**
  - Das vorhandene DOM wird zerstört
  - Request wird an den Server geschickt
  - Keine Aktionen möglich, während der Request läuft
  - Warten auf Response (meist auf leerer Seite)
- **Erst wenn die Response da ist, ist die Anwendung wieder nutzbar**
  - UI wird immer komplett neu gerendert
  - Server muss den App-State rekonstruieren (z.B. Sub-Menu)
  - Lange Wartzeit bei hoher Serverlast

Traditionelle Web Anwendungen

- **Zum Teil sind Workarounds notwendig**
  - Redirect-after-post um mehrfaches absenden von Formularen durch Neuladen zu vermeiden
- **Um Neuigkeiten zu sehen, muss immer alles neu geholt werden**

AJAX

Asynchronous JavaScript and XML (AJAX)

- **Asynchronous**
  - I/O Operation werden an das Betriebssystem delegiert
  - Der eigene Kontrollfluß läuft **sofort** weiter
  - Das Betriebssystem meldet zurück, wenn die Operation abgeschlossen ist
  - Beispiele: Boost Asio, AsyncIO, node.js
- **JavaScript**
  - Die verwendete Sprache
- **XML**
  - Tatsächlich kann alles via XHR transportiert werden, nicht nur XML (JSON, Bilder, Filme, HTML, etc.)

Sequentielle vs. Asynchrone Programmierung

- **Sequentiell:**
  - Der Code wird Zeile für Zeile (bzw. Statement für Statement) von oben nach unten durchlaufen

```
1: console.log("Hallo Welt!");
2: console.log("Tschüss!");

// Ausgabe:
Hallo Welt!
Tschüss!
```

Sequentielle vs. Asynchrone Programmierung

- **Asynchron:**
  - Eine Aktion wird gestartet, der Kontrollfluss kehrt sofort zum Aufrufer zurück
  - Zu einem späteren Zeitpunkt wird dann die gewünschte Aktion ausgeführt

```
1: let lambda = () => console.log("Hallo Welt!");
2: window.setTimeout(lambda, 1000);
3: console.log("Tschüss!");

// Ausgabe:
Tschüss!
Hallo Welt!
```

Problem: Was wenn ich zu spät bin?

- **Asynchrone Programmierung erfordert immer die Angabe eines Callbacks**
  - Was wenn der Callback erst gesetzt wird, wenn die asynchrone Operation schon abgeschlossen ist?

```
// Falsch: Erst schicken, dann registrieren
let xhr = new XMLHttpRequest();
xhr.open("GET", "http://www.example.org/example.txt");
xhr.send();
// Zeit vergeht...
xhr.addEventListener("load", () => console.log("Done"));
```

  - Ergebnis: Unser Code wird nie ausgeführt, weil wir ihn zu spät registrieren.

Problem: Was wenn ich zu spät bin?

- **Asynchrone Programmierung erfordert immer die Angabe eines Callbacks**
  - Erst den Callback setzen, dann den Request abschicken.

```
// Richtig: Erst registrieren, dann schicken
let xhr = new XMLHttpRequest();
xhr.addEventListener("load", () => console.log("Done");
xhr.open("GET", "http://www.example.org/example.txt");
xhr.send();
```

  - Ergebnis: Wenn der Request abgeschlossen ist, wird unser Code ausgeführt.

Promises / Futures / Deferred

- **Ein Promise kapselt eine asynchrone Operation**
- **Es liefert APIs um eigenen Code auszuführen wenn die asynchrone Operation:**
  - Erfolgreich war (done)
  - Nicht erfolgreich war (fail)
  - In Beiden fällen (any)
  - Eine Exception auftrat (catch)
- **Alle jQuery AJAX Funktionen liefern ein Promise**
- **jQuery Deferred Object:**
  - https://api.jquery.com/category/deferred-object/

Promises / Futures / Deferred

- Ein Promise garantiert, dass auch dann der Callback aufgerufen wird, wenn man sich zu spät registriert!

```
let promise = $.get("index.html");
// 10 Minuten später
promise.done(() => console.log("Fertig"));


// Ausgabe:
Fertig
```

Nice to know: Closures!

- Innerhalb von Lambdas werden alle lokalen Variablen aus dem umgebenden Scope kopiert
- Praktisch um später den „state“ zu tracken

```
function closureDemo() {
   let now = new Date().toString();
   let lambda = () => console.log("[Lambda] The Time: " + now);
   window.setTimeout(lambda, 2000);
   console.log("[Function] The Time: " + now);
}
// Ausgabe:
[Function] The Time: Sun Apr 28 2019 19:00:57
[Lambda] The Time: Sun Apr 28 2019 19:00:57
```

Fetch API für HTTP Requests

- **Nachfolger von XMLHttpRequests (XHR)**
  - Ermöglicht das asynchrone verschicken und empfangen von HTTP-Requests
  - Ersetzt XHR in mordernen Browser
    https://caniuse.com/#feat=fetch
- **Vollständig asynchrone API basierend auf Promises**
- **Details:**
  https://developer.mozilla.org/de/docs/Web/API/Fetch_API

Fetch Promises Beispiel

```
// 1) Response Body in Console loggen
let promise = fetch("https://google.de/);
promise.then((response) => console.log(response.text()));



// 2) JSON Extrahieren
// data.json: [ { "name": "Guybrush" }, { "name": "Thomas" } ]
let promise = fetch("http://some.api.de/data.json");
promise.then((response) => {
    return response.json();
})
.then((jsonData) => {
    alert("Der Name ist: " + jsonData[0].name);
});
```

Fetch async/await Beispiel

- async/await ist „syntactical sugar“ um promises nicht manuell „auspacken“ zu müssen
- Durch async/await sieht asynchroner Code aus, wie sequentieller Code (ist aber weiterhin asynchron!)

```
// data.json: [ { "name": "Guybrush" }, { "name": "Thomas" } ]
async function getName() {
   let response = await fetch("http://some.api.de/data.json");
   let jsonData = response.json();
   alert("Der Name ist: " + jsonData[0].name);
}
```

jQuery Funktionen für AJAX

- **$.get(URL, DATA)**
  - Schickt einen HTTP-GET Request.
  - DATA kann ein JSON-Objekt sein, dass dann zur Get-Parameterns serialisiert wird
- **$.getJSON(URL, DATA)**
  - Genau wie $.get, erwartet als Antwort aber JSON
- **$.post(URL, DATA)**
  - Schickt einen HTTP-Post-Request
  - DATA kann ein JSON-Objekt sein, dass dann zu einem application/x-www-form-urlencoded HTTP-Body serialisiert wird
- **https://api.jquery.com/category/ajax/shorthand-methods/**

Beispiel: „guestbook-api“

- Beim anlegen eines neuen Beitrages, werden nur die Einträge neu geladen
- Beim löschen eines Eintrages, verschwindet dieser
- Die Seite wird durch normale Benutzeroperationen nie neu geladen
- Siehe im Repo unter:
  - Beispiele/Node.JS/guestbook-api/

Weiterführende Links

- Weiterführende Links für asynchrone Programmierung
  - https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous
  - https://www.youtube.com/watch?v=PoRJizFvM7s
  - https://www.w3schools.com/js/js_callback.asp