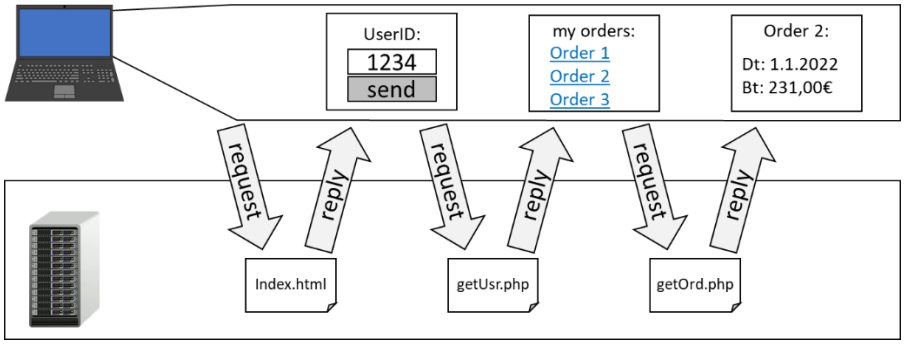
1 Synchroner Austausch mit dem Server

Die ursprüngliche Idee der Webkommunikation mit einem Applicationserver war, dass man einen http-Request gesendet hat, dieser vom Server verarbeitet wird und er als Response eine komplett neue HTML-Seite sendet. Dadurch musste clientseitig die angezeigte Seite auch nach jeder Kommunikation neu aufgebaut werden.



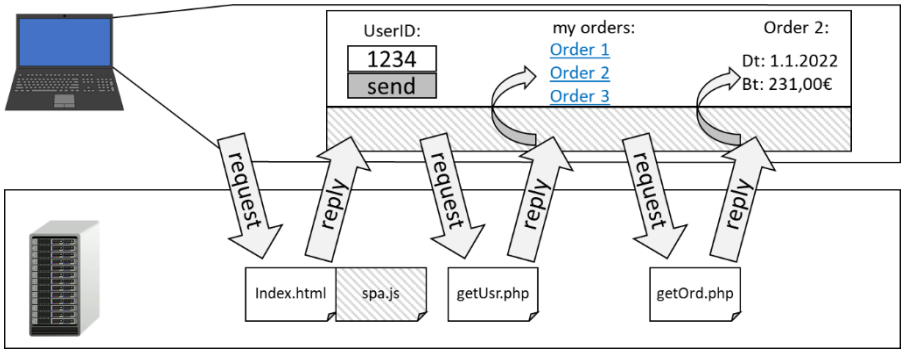
Dies hatte zwei wesentliche Nachteile:

- Es gab keinen lückenlosen Arbeitsablauf am Client – man musste immer auf den Seitenaufbau nach einer Serverkommunikation warten
- Es mussten Daten immer neu gesendet werden, welche eigentlich dem Client schon bekannt waren. Caching hat dieses Problem lediglich bei statischen Daten behoben.

Es musste also eine Alternative gefunden werden.

2 Asynchroner Austausch mit dem Server

Die Idee war, dass man mit CSR die Seite dynamisch aufbaut und im Hintergrund die Datenkommunikation asynchron mit dem Server durchführt.



Anfangs war dies nur über versteckte Frames möglich. Ab 2006 wurde das XMLHttpRequest Objekt in JavaScript eingeführt. Wie der Name schon sagt, hat man auf XML für die Formatierung der Informationen gesetzt. Zu dieser Zeit hat sich auch der Begriff AJAX (Asynchronous JavaScript and XML) etabliert. Sehr früh hat man aber die Vorteile von JSON bezüglich des geringeren Overheads gesehen und setzt nun fast ausschließlich auf JSON. Später wurde fetch eingeführt, welches XMLHttpRequest langsam, aber sicher ersetzt. Wir werden uns hier jedoch beide Optionen kurz ansehen.

Beide Verfahren erlauben es uns, direkt mit JavaScript einen http-Request an einen Server zu senden und den Reply zu verarbeiten, ohne dass die Seite neu geladen werden muss.

3 Kommunikation mit XMLHttpReqest

Der Name XMLHttpRequest ist (leider) etwas irreführend, weil faktisch kein XML gesendet wird, sondern lediglich ein beliebiger String – meist JSON. Hier eine beispielhafte Implementierung eines http-Requests:

```
function doRequest(httpMethod, url, myData, callback){
  const xhr = new XMLHttpRequest();
  xhr.open(httpMethod, url, true);
  xhr.onreadystatechange = () => {
    if (xhr.readyState == 4) {
      if (xhr.status == 200){
        callback(JSON.parse(xhr.response));
      } else {
        alert("error:" + xhr.status);
      }
    }
  }
  xhr.setRequestHeader("Content-Type",
     "application/json;charset=UTF-8" );
  xhr.send(JSON.stringify(myData));
}
function processData(data){
    alert("st.: " + data.replyMessage + "\nDat: " + data.replyDetails);
}
function doSendMyData() {
  const myData = {myName:"John Doe"};
  doRequest("POST", "http://localhost:8080/api/myData",
     myData, processData);
}
```

Wir beginnen mit der Funktion „`doRequest`“. Ich habe hier vier Parameter eingebaut:

- `httpMethod` – Hier kommt die http Methode rein (bspw. `GET` oder `POST`).
- `url` – die Webadresse des zu verarbeitenden Serverelements (in unserem Fall also ein Endpoint auf dem Localhost). Achtung – die URL muss im Normalfall vom gleichen Server stammen, wie die HTML-Seite, damit die “Same-origin policy“ greift[1]
- `myData` – das zu sendende JavaScript Objekt.
- `callback` – dies ist der Funktionsname der Funktion, welche die Verarbeitung der gesendeten Daten übernehmen soll. Oft wird hier auch einfach nur eine anonyme Funktion (also mit dem Arrow Operator) eingetragen

Zuerst erzeugen wir ein `XMLHttpRequest` Kommunikationsobjekt. Die Verbindung wird mit „`open`“ geöffnet, wobei wir hier zum einen die Kommunikationsmethode, dann die URL der verarbeitenden Serverseite und das Flag „`true`“ für asynchrone Verarbeitung setzen. In unserem Beispiel verwenden wir „`POST`“, da wir hier auch das Senden von Daten im Body sehen. Nun müssen wir noch den „`content tpye`“ im Header setzten, welchen wir für unser Beispiel auf JSON setzen. Die eigentlichen Daten setzen wir dann später in der „`send`“ Methode, in der wir unser zu sendendes JavaScript Objekt in einen JSON-String umwandeln.

Danach hängen wir eine Methode an das Objekt, welche immer dann aufgerufen wird, wenn der Verarbeitungsstatus sich ändert (`onreadystatechange`). Diese soll lediglich prüfen, ob der Verarbeitungsstatus die 4 ist (also „done“). Erst dann sind alle Daten gesendet worden. Der HTML Satuscode muss die 200 („ok“) sein, ansonsten geben wir eine Fehlermeldung aus. An dieser Stelle verzichten wir auf ein weitergehendes Errorhandling. Wenn die beiden Bedingungen erfüllt sind, rufen wir die übergebene Callback Methode mit den empfangenen Daten auf. Wenn die Verarbeitungsmethode definiert wurde, schicken wir den Request mit „`send`“ ab.

Für den Callback haben wir die Funktion „`processData`“ erstellt, welche lediglich die Daten als Popup anzeigt. Der Aufruf in „`doSendMyData`“, übergibt jetzt die einzelnen Daten und stößt den Prozess an.

[1] Wenn diese aufgeweicht werden soll, siehe https://expressjs.com/en/resources/middleware/cors.html

4 Kommunikation mit fetch()

Fetch wurde eingeführt, um den Sendeprozess schlanker zu gestalten und einige Defizite des `XMLHttpRequest` Ansatzes zu umgehen. Hierfür wurden relativ viele Möglichkeiten zur Konfiguration dieses Ansatzes vorgesehen[2]. Trotzdem noch einige wenige Features des `XMLHttpRequest` Objektes nicht umgesetzt wurden ist davon auszugehen, dass dies in naher Zukunft passieren wird. Für die meisten Anwendungsfälle wird entsprechend empfohlen, den `fetch()` Ansatz zu nutzen. Im folgenden Code ist die gleiche Funktionalität wir aus dem vorausgegangenen Kapitel als `fetch()` umgesetzt:

```
async function doFetch(httpMethod, url, myData) {
  const response = await fetch(url, {
  method: httpMethod,
  headers: {'Content-Type': 'application/json;charset=UTF-8'},
    body: JSON.stringify(myData)
  });
  const data = await response.json();
  return data;
}
function doSendMyData() {
  const myData = {myName:"John Doe"};
  doFetch("POST", "http://localhost:8080/api/myData", myData)
  .then((data) => {
    processData(data);
  })
  .catch((error) => {
    alert("error:" + error);
  });
}
```

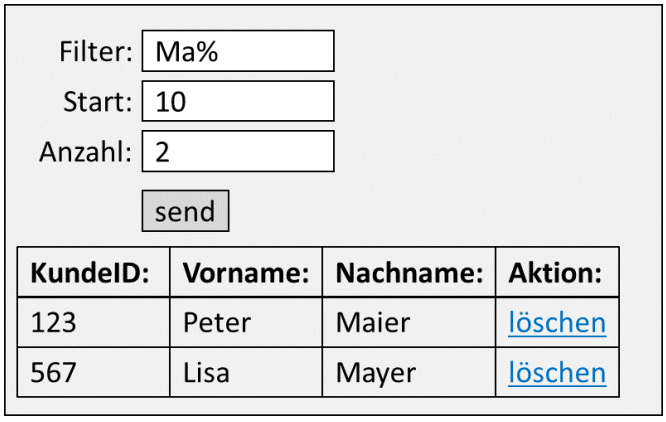
Wie man sieht, hat man bei der Implementierung von `fetch()` auf Promises gesetzt, weshalb wir einen `async-await` Aufruf realisieren können. Dadurch ist die gesamte Nutzung relativ übersichtlich gehalten. Das `response` Objekt, welches von `fetch()` zurückgegeben wird, bringt auch eine Methode `json()` mit. Dies wandelt den im Body hinterlegten JSON-String in ein JavaScript Objekt um – gibt aber das Ergebnis als Promise zurück. Die hier dargestellte Nutzung von `fetch()` zeigt allerdings einen Minimalansatz. Wer weitere Funktionen nutzen möchte, muss im unten eingefügten Link (oder ähnlichen online Ressourcen) weitere Details einholen.

5 Aufgabenstellung

Schreiben Sie eine HTML-Seite als SPA, welche die REST-Schnittstelle aus der vorausgegangenen Übung nutzt und umsetzt. Beschränken Sie sich hier aber auf die folgenden beiden Use-Cases:

`GET: api/customer`
`DELETE: api/customer/`*id*

Das hierfür notwendige Userinterface sollte nebenstehende Elemente enthalten.



[2] Details: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch