Ajax, XHR, HTTP a jejich kamarádi

- **AJAX** = Asynchronous JavaScript and XML
  - požadavky nemusí být asynchronní
  - přenášet se dá cokoliv (nejen XML)

XHR a HTTP

- XMLHttpRequest je alfou a omegou pro bohaté webové aplikace
- možnost provést HTTP požadavek
- dočítání dat, odesílání informací
- v moderních prohlížečích není nutná žádná abstrakce

```
1. var xhr = new XMLHttpRequest();
2.
3. xhr.addEventListener("readystatechange", /* ... */);
4. xhr.addEventListener("load", /* ... */);
5.
6. xhr.open(metoda, url, async);
7. xhr.send(data);
```

- lze použít jakoukoliv HTTP metodu
- synchronní požadavek blokuje vykreslovací vlákno
- URL požadavku podstupuje kontrolu originu
- použití **onload** vs. **addEventListener** jako u běžných DOM událostí

- **doplňková funkcionalita:**
  - setRequestHeader(name, value)
  - getResponseHeader(name), getAllResponseHeaders()
  - abort()
  - overrideMimeType()

- **vlastnosti a data odpovědi:**
  - **xhr.readyState** = stav požadavku (4 = hotovo)
  - **xhr.status** = HTTP kód odpovědi (200 = OK)
  - **xhr.responseText** = data odpovědi
  - **xhr.responseXML** = data odpovědi ve formátu XML (je-li k dispozici)

XMLHttpRequest 2

- rozšíření se zpětně kompatibilním API
- události progress, load, error, abort
- to samé na objektu xhr.upload
- xhr.responseType = „arraybuffer“, „blob“, „document“, „json“, „text“
- xhr.response

Sandbox, Origin a CORS

- XHR nelze poslat na jakékoliv URL (kvůli bezpečnosti)
    - součástí požadavku jsou cookies, HTTP autorizace, certifikáty,…

- cross-damin požadavky – potencionální riziko

```
1. var xhr = new XMLHttpRequest();
2. xhr.open("get", "http://gmail.com/", true);
3. xhr.send();
4.
5. xhr.onload = function() {
6.     alert(this.responseText);
7. }
```

- **Sandbox**: omezení originem
    - XHR jsou v základu omezeny jen na URL se stejným originem
    - origin = schema + host + port

CORS (Cross Origin Resource Sharing)

- oslabuje sandbox tam, kde to nepředstavuje riziko
- **myšlenka**: zeptat se backendu, jestli je ochoten takovýto požadavek přijmout
    - pokud ne (default), odpověď se zahodí
    - pokud ano, odpověď se vrátí

- **implementace**:
    - HTTP hlavička požadavku **Origin**
    - podpora inzerovaná v hlavičce odpovědi **Access-Control-Allow-Origin**
    - požadavek se vždy provede
    - u složitějších požadavků (file upload, DELETE,…) je to složitější
    - nouzové řešení je server-side proxy (stejný origin)

Transportní formáty

- data lze posílat v mnoha formátech
- žádný není nejlepší
- upload zpravidla neřešíme, download dle potřeby
- plain text – to není žádný formát

XML

- odpověď musí být validní XML
- odpověď musí mít platný XML mime type
- xhr.responseXML instanceof DOMDocument
- dotazování přes DOM nebo Xpath

JSON

- textový formát, podmnožina JS
- řetězce, čísla, bool, pole, struktury, null
- absence speciálních číselných hodnot, undefined, Date, RegExp

```
{
    "name": "jan",
    "data": [3, 4, true]
}
```

- klíče musí být ohraničeny uvozovkami
- JSON.parse, JSON.stringify
- eval jako zpětná kompatibilita

Binární

- lze, ale obtížné (cross-browser)
- u XMLHttpRequest 2 lze přes responseType a send(binaryData)
- v JS dlouho nebyl vhodný datový typ pro práci s binárními daty

JSONP

- **NENÍ** transportní formát
- úplně jiná technika přenosu dat
- způsob obcházení cross-origin restriction

```
<!-- vložit do dokumentu -->
<script src="http://api.com/service?callback=mojeFunkce"></script>
```

```
/* odpověď serveru */
mojeFunkce({name: "jan", data: [3, 4, true]});
```

fetch()

- nové, nedávno standardizované rozhraní
- podobné jako XMLHttpRequest, ale jednodušší
- asynchronní řízení pomocí Promises

```
fetch("/nejaky/soubor.json")
    .then(function(response) {
        if (response.status != 200) {
            console.log("HTTP status", response.status);
            return;
        }

        response.json().then(function(data) {
            console.log(data);
        });
    })
    .catch(function(err) {
        console.log("Error", err);
    });
```

- **fetch() vs. XHR**
  - chybí timeout → lze implementovat ručně pomocí setTimeout a abort()
  - chybí abort() → problém celého konceptu Promises
  - objekt AbortController má výhledově řešit přerušitelné Promises

Relevantní drobnosti

- prohlížeče omezují počet současných XHR na stránku (zpravidla 4)
- HTTP požadavek musí být iniciován klientem
- technika long poll / comet: pošlu HTTP požadavek; server odpoví, až bude mít data
- server takto může notifikovat, ale musí držet mnoho otevřených spojení

Web Sockets

- snaha o zpřístupnění duplexní komunikace v prohlížeči
- notifikace ze strany serveru
- persistentní spojení (trvale otevřené spojení na server)
- možnost oboustranného posílání dat
- vlastní protokol postavený nad HTTP Upgrade
- žádná spojitost s BSD sockety

Klientská část

```
1. var socket = new WebSocket("ws://server.tld:1234/");
2.
3. socket.onopen = function(e) {
4.     socket.send("data");
5. }
6.
7. socket.onmessage = function(e) {
8.     alert(e.data);
9.     socket.close();
10. }
```

Serverová část

- Ta nás nezajímá!
- nutný specializovaný SW, knihovny pro řadu jazyků

Protokol

- vlastní binární protokol
- komunikace začíná HTTP požadavkem s hlavičkou Upgrade
- zpětná kompatibilita s existujícími webovými servery