

Univ. Babeş-Bolyai,
Facultatea de Matematică şi Informatică
Lect. dr. Darius Bufnea
Notiţe de curs Programare Web: AJAX

Pe lângă prezentul material, vă rog de asemenea „ferm” (lectură obligatorie) să studiaţi şi următoarele materiale:

[AJAX Tutorial](#)
[jQuery – AJAX Introduction](#)
[jQuery AJAX Methods](#)

De la ce vine? Ce face?

AJAX este abreviere de la **A**synchronous **J**avaScript and **X**ML. După cum îi spune şi numele, este o „tehnologie” (poate cam mult spus..., mai degrabă o „tehnică”) de încărcare asincronă în JavaScript de conţinut care se dorea iniţial a fi XML, însă în prezent conţinutul respectiv este de obicei o expresie JSON, dar poate fi orice tip de conţinut: HTML, „raw” data (data brute care să conţină de exemplu un stream video).

Mai simplu spus, prin execuţia de cod JavaScript, permite încărcarea de nou conţinut adus de pe server, încărcare care se face „în spate” („în spate” = „în background” = „asincron”) prin intermediul unui nou request HTTP făcut la serverul web şi transparent pentru utilizatorul care vizualizează pagina. Acest conţinut este adus fără reîncărcarea completă a documentului curent afişat în browser. De obicei pe baza conţinutului nou adus de pe server, din JavaScript se actualizează doar o parte a documentului (a DOM-ului) modificându-se valoarea unor anumite elemente HTML din documentul curent încărcat sau creându-se elemente HTML noi.

Foarte important: Prin intermediul unui apel AJAX se primeşte nou conţinut de la serverul web, dar în acelaşi timp se poate şi trimite conţinut dinspre client (browser) spre serverul web.

Până la apariţia AJAX, singura modalitate de comunicare şi de trimitere de date între browser şi serverul web se făcea trimiţând „full” request-uri de la browser la server. Aceste request-uri (de exemplu un request GET făcut printr-un click simplu pe un link – ancora <a> sau un submit prin GET sau POST la un formular) redirectau întotdeauna browser-ul la un nou URL, în pagina fiind încărcat un nou document, browser-ul fiind nevoit să construiască un nou DOM pentru acesta. Folosind AJAX, documentul afişat nu se mai reconstruieşte/reîncarcă complet, ci doar parţial.

IMPORTANT: ÎN URMA UNUL APEL AJAX, URL-UL DOCUMENTULUI CURENT ÎNCĂRCAT NU SE SCHIMBĂ (DOCUMENTUL RĂMÂNE ACELAŞI). ÎN URMA UNUI SUBMIT „CLASIC”, URL-UL SE SCHIMBA, PAGINA ÎNCĂRCATĂ ESTE UNA NOUĂ (CU TOTUL ALTA), BROWSER-UL CONSTRUIND UN NOU DOM.

Exemplu de apel clasic vs. apel AJAX

Enunț de problemă (simplă): Să se trimită de pe front-end (client) un șir la server-ul web, serverul web să convertească acest text la majuscule și să trimită textul cu litere mari înapoi clientului (browser-ului care sa-l afișeze).

Fără a insista pe partea de back-end, și nici pe rezolvările în sine în acest moment, prezentăm mai jos variatele de rezolvare, interesându-ne mai mult **comportamentul** diferit al acestora:

Varianta 1 (disponibila online [aici](#)) – presupune submit „clasic” al unui formular și trimiterea textului prin una dintre metodele GET sau POST. Adresa scriptului care se execută pe back-end devine noul URL activ încărcat în browser, output-ul acestui script devenind noul document încărcat în browser. Scriptul care se execută pe back-end primește datele de la formular, convertește textul la majuscule și afișează un nou document HTML care conține și textul care se dorește a fi convertit la majuscule. Avantajul acestei abordări este ca nu folosește nicio tehnologie care se execută pe client (JavaScript).

Front-end (fișier index.html)

```
<!DOCTYPE html>
<html lang="ro-RO">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" >
  <title>Submit clasic</title>
</head>
<body>
<form action="toUpper.php" method="GET">
Introduceti un sir: <input type="text" name="sir"><br>
<input type="submit" value="Trimite">
</form>
</body>
</html>
```

Back-end (toUpper.php)

```
<!DOCTYPE html>
<html lang="ro-RO">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>toUpper clasic</title>
</head>
<body>
Sirul primit de la client convertit la majuscule este:
<?php
  echo strtoupper($_GET["sir"]);
?>
</body>
</html>
```

Observați pe varianta 1 de mai sus faptul că la submit-ul formularului, URL-ul documentului deschis în browser se schimbă (din index.html în toUpper.php), browser-ul încărcând practic o nouă pagină și construind un nou DOM.

Varianta 2 (disponibila online [aici](#)) – presupune trimiterea textului care se dorește convertit la majuscule printr-un call (apel) AJAX. Acest apel se face tot printr-una dintre metodele GET sau POST însă apelul se face “în background” (asincron). Pagina încărcată în browser nu se schimbă în timpul rulării exemplului (URL-ul acesteia rămânând același). Conversia textului la majuscule se face tot pe back-end prin execuția unui script, însă outputul acestui script nu se va afișa în mod direct în browser (nu trebuie să fie un document complet HTML pentru care browser-ul să construiască un nou DOM) ci se trimite ca răspuns apelului AJAX fiind ulterior prelucrat în JavaScript. În DOM-ul documentului inițial încărcat se vor face modificări minimale (spre exemplu afișarea textului primit convertit la majuscule). Față de prima variantă, avantajul acestei abordări este o mai mare interactivitate cu utilizatorul, un UI (User Interface) cu un comportament mai “user friendly”, și posibilitatea apelării back-end-ului și de către clienți non web-based (spre exemplu clienți mobile sau desktop). Dezavantajul este un mai mare efort computațional pe client (browser-ul are de executat mai multe „lucruri”, lucru care se face pe cheltuiala - CPU, memorie, wați consumați - utilizatorului).

Front-end (fișier index.html)

```
<!DOCTYPE html>
<html lang="ro-RO">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>Call AJAX</title>
</head>
<body>
  Introduceti un sir: <input type="text" id="sir" onkeyup="doAJAXRequest()"><br>
  Sirul trimis la server si primit inapoi convertit la majuscule este:
  <span id="rezultat"></span>
</form>
<script type="text/javascript">

function doAJAXRequest() {
  var request = new XMLHttpRequest();

  request.onreadystatechange = function() {
    if (request.readyState == 4) { // cerere rezolvata
      if (request.status == 200) // raspuns Ok
        document.getElementById('rezultat').innerHTML = request.responseText;
      else
        console.log('Eroare request.status: ' + request.status);
    }
  };

  request.open('GET', 'toUpper.php?sir=' + document.getElementById('sir').value,
true);
  request.send('');
}

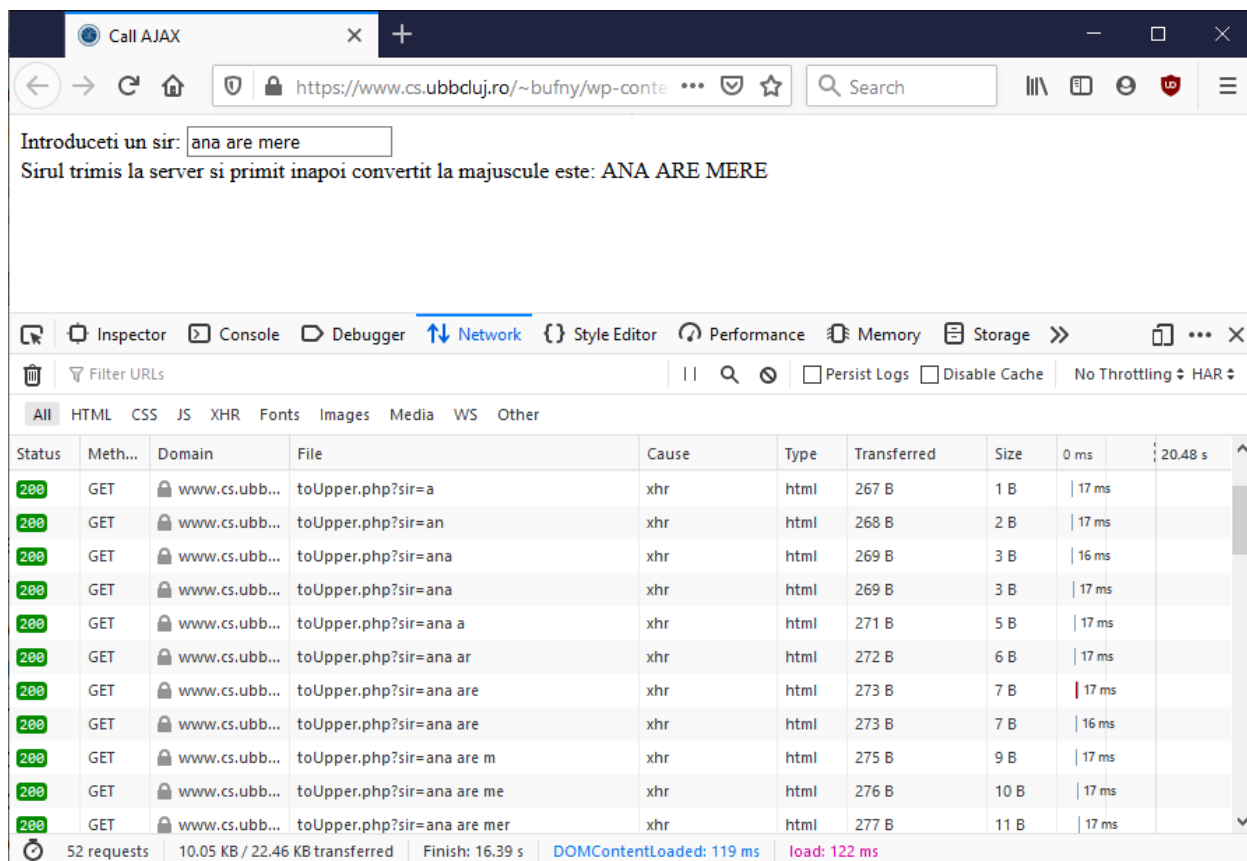
</script>
</body>
</html>
```

Back-end (toUpper.php)

```
<?php
  echo strtoupper($_GET["sir"]);
?>
```

Observați pe varianta 2 de mai sus faptul ca URL-ul documentului deschis în browser nu se schimbă, browser-ul operând toate modificările în DOM-ul același pagini.

Este util și didactic în acest context să observați în Developer Tools și request-urile AJAX care se fac. Cu Developer Tools-ul pornit (F12), selectați tab-ul Network (“Rețea” pentru fanii “Decupează și Lipește” ☺), și introduceți câteva caractere în input-ul de tip text pentru a observa apelurile AJAX efectuate la apariția evenimentului `onkeyup`. Acest comportament este surprins în captura de mai jos (Firefox, dar comportamentul este similar și în Google Chrome).



The screenshot shows a web browser window with the address bar displaying `https://www.cs.ubbcluj.ro/~bufny/wp-conte...`. The page content includes a text input field with the text "ana are mere" and a message below it: "Sirul trimis la server si primit inapoi convertit la majuscule este: ANA ARE MERE". The Developer Tools window is open, showing the Network tab. The list of requests shows a series of GET requests to `toUpper.php?sir=` followed by the characters of the input string. The status bar at the bottom indicates 52 requests, 10.05 KB / 22.46 KB transferred, and a finish time of 16.39 s.

Status	Meth...	Domain	File	Cause	Type	Transferred	Size	0 ms	20.48 s
200	GET	www.cs.ubb...	toUpper.php?sir=a	xhr	html	267 B	1 B	17 ms	
200	GET	www.cs.ubb...	toUpper.php?sir=an	xhr	html	268 B	2 B	17 ms	
200	GET	www.cs.ubb...	toUpper.php?sir=ana	xhr	html	269 B	3 B	16 ms	
200	GET	www.cs.ubb...	toUpper.php?sir=ana	xhr	html	269 B	3 B	17 ms	
200	GET	www.cs.ubb...	toUpper.php?sir=ana a	xhr	html	271 B	5 B	17 ms	
200	GET	www.cs.ubb...	toUpper.php?sir=ana ar	xhr	html	272 B	6 B	17 ms	
200	GET	www.cs.ubb...	toUpper.php?sir=ana are	xhr	html	273 B	7 B	17 ms	
200	GET	www.cs.ubb...	toUpper.php?sir=ana are	xhr	html	273 B	7 B	16 ms	
200	GET	www.cs.ubb...	toUpper.php?sir=ana are m	xhr	html	275 B	9 B	17 ms	
200	GET	www.cs.ubb...	toUpper.php?sir=ana are me	xhr	html	276 B	10 B	17 ms	
200	GET	www.cs.ubb...	toUpper.php?sir=ana are mer	xhr	html	277 B	11 B	17 ms	

52 requests | 10.05 KB / 22.46 KB transferred | Finish: 16.39 s | DOMContentLoaded: 119 ms | load: 122 ms

Observații:

- Pentru a executa exemplele de mai sus este nevoie de execuția unui cod minimal pe back-end. Pentru execuția acestuia este nevoie de un server web capabil să "înțeleagă"/execute tehnologia (sau să delege mai departe execuția) în care e scris back-end-ul. Logica din exemplele de față care fac conversia la majuscule este scrisă în PHP – dar mai multe în acest sens în cursul următor.
- Nu este nicio legătură directă între AJAX și PHP, cele două tehnologii nefiind dependente una de cealaltă. Un apel AJAX are nevoie de un end-point care să se execute pe server, din rațiuni didactice multe exemple din prezentul material având partea de back-end scrisă în PHP (dar sunt și unele exemple în care back-end-ul e scris în C sau în shell UNIX).

Ce este în spate la apel AJAX?

În „spatele” unui apel AJAX stă un obiect numit `XMLHttpRequest`, de fapt o instanță a acestui obiect (`XMLHttpRequest` este funcție membră pe obiectul `Window`). Pentru a face un apel AJAX, trebuie construit în JavaScript un astfel de obiect:

```
var request = new XMLHttpRequest();
```

variabila `request` abstractizând toată cererea AJAX care se dorește a fi făcută. Acest obiect prezintă o serie de metode și stări (în funcție de starea apelului AJAX și starea comunicării cu serverul web).

Metode importante pe obiectul `XMLHttpRequest` (pe o instanță de acest tip):

`open(metoda, URL, async)`

Specifică URL spre care se va face apelul AJAX (fără a se iniția efectiv acest apel). Semnificația parametrilor este următoarea:

- `metoda`: metodă HTTP (GET, POST, etc.) ;
- `URL`: URL-ul spre care se face apelul AJAX, poate fi un URL relativ sau absolut (atenție, în cazul URL-urilor absolute intervine în calcul și ceea ce se cheamă Same Origin Policy)
- `async`: boolean, recomandabil setat la `true`.

Observație: există mai multe forme de `open`, unele permit inclusiv trimiterea unui nume de utilizator și a unei parole dacă URL-ul invocat necesită autentificare.

`send(content)`

Trimite efectiv cererea AJAX (pe obiectul AJAX trebuie să se fi executat anterior `open`). Dacă cererea se face prin GET, parametrul `content` este vid. Recapitulare de la cursul / laboratorul de HTTP: o cerere făcută prin GET nu avea conținut după `new` line-ul de după header-ele (antetele request-ului). Acest conținut (care în cazul de față se specifică prin parametrul `content`) era prezent doar în cazul unui apel făcut prin POST.

Observație: Dacă apelul AJAX se face prin GET parametrul `content` poate să lipsească - unele versiuni de Internet Explorer cer însă și în acest caz specificarea șirului vid `""` ca parametru actual.

Modul de încapsulare a datelor trimise prin POST de la client la server sau de la server la client depinde mult de tehnologia folosită pe partea de back-end. Astfel, datele trimise de la client la server pot fi trimise folosind mai multe „încapsulări”. Un exemplu este încapsularea clasică „`atribut1=valoare1&atribut2=valoare2&atribut3=valoare3...`” folosită de protocolul HTTP la submiterea prin GET sau POST a unui formular (ocazie bună să recapitulați ceea ce înseamnă `QUERY_STRING` de la cursul/laboratorul de HTTP). Datele de la client la server (și invers) pot fi trimise și în format JSON, XML, sau propria încapsulare a programatorului (dacă aceste dorește „să se lege la cap” cu propria parsare ☺) a datelor atât pe partea de front-end cât și pe partea de back-end).

Dacă apelul AJAX se face prin GET, `send` se apelează cu șirul vid ca parametru. Și în acest caz însă, clientul (browser-ul) poate trimite date la server specificând aceste date sub forma unui `QUERY_STRING` de forma `?atribut1=valoare1&atribut2=valoare2&atribut3=valoare3` specificat în URL dat

ca parametru metodei `open` anterioare (recapitulare de la HTTP: datele trimise de la browser la server în urma unui submit de formular făcut prin GET ajungeau în `QUERY_STRING`-ul scriptului specificat ca și valoare pentru atributul `action` al formularului).

Alte metode utile apelabile pe un obiect request AJAX instanță a obiectului `XMLHttpRequest`:

Metoda	Descriere
<code>setRequestHeader(header, value)</code>	Adaugă un nou antet HTTP (header) la cererea care va fi trimisa la serverul web
<code>getResponseHeader(header)</code>	Întoarce valoarea header-ului HTTP specificat de pe răspunsul pe care serverul web îl dă în urma apelului AJAX
<code>getAllResponseHeaders()</code>	Întoarce toate headere-le HTTP de pe răspuns

Pe lângă metodele de mai sus, obiectul `XMLHttpRequest` prezintă și o serie de date membre/proprietăți importante, cele mai importante dintre acestea fiind `onreadystatechange`, `readyState`, `responseText` și `status`.

Prin intermediul proprietății `readyState` se poate verifica starea apelului AJAX și a comunicării dintre browser și server-ul web. Această proprietate poate lua următoarele valori:

- 0 – apelul AJAX este neinițializat, s-a construit obiectul, dar nu s-a efectuat `open`;
- 1 – s-a efectuat `open`, dar nu s-a făcut încă `send`;
- 2 – cererea AJAX este trimisă, s-a efectuat `send`, dar încă nu a sosit răspunsul;
- 3 – apelul AJAX este în starea receiving și datele continuă să sosească, răspunsul de la server nu este complet;
- 4 – apelul AJAX s-a terminat, a sosit tot răspunsul de la server.

Apelul AJAX se consideră a fi terminat cu succes când `readyState`-ul său este 4 și `status`-ul său este 200. Proprietate `status` a obiectului `XMLHttpRequest` va conține în urma apelului codul de răspuns trimis de serverul web prin intermediul protocolului HTTP (ocazie bună să le recapituți: 200 – OK, 404 – Not Found, 403 – Forbidden, 500 – Internal Server Error, etc.). De asemenea proprietatea `statusText` a obiectului `XMLHttpRequest` va conține mesajul „human readable” asociat codului de răspuns HTTP (OK, Not Found, Forbidden, etc.).

Poate ce mai importantă proprietate a obiectului `XMLHttpRequest` este `onreadystatechange`. Prin intermediul acestei proprietăți se poate specifica o funcție (care poate fi și o funcție anonimă JavaScript) care să se execute (după cum spune și numele proprietății) atunci când obiectul AJAX își schimbă starea (`readyState`-ul) din 0 în 1, din 1 în 2, ș.a.m.d. Practic funcția specificată ca valoare pentru proprietatea `onreadystatechange` se apelează de mai multe ori, însă cel mai util apel al său este când apelul AJAX este în starea 4 (s-a terminat), stare în care de obicei se verifică și `status`-ul cu care s-a terminat apelul (ideal 200).

Atribuirea unei valori pentru proprietatea `onreadystatechange` este oarecum obligatorie, fără specificarea unei funcții prin intermediul acestei proprietăți apelul AJAX deși se efectuează își pierde „esența”. La terminarea cu succes a apelului AJAX (când `readyState`-ul este 4 și `status`-ul 200 - și numai atunci!) proprietatea `responseText` a obiectului `XMLHttpRequest` conține tot răspunsul oferit de serverul web în urma apelului AJAX. Acest răspuns poate fi fie plain text ca în exemplele de mai sus, fie o expresie JSON după cum vom folosi într-un exemplu viitor, fie XML, etc.

Observație: Pe versiunile mai vechi de Internet Explorer, obiectul `request` care stă în spatele unui apel AJAX trebuie construit în modul următor, restul funcționalităților acestuia rămânând identice:

```
request = new ActiveXObject('MSXML2.XMLHTTP');
```

sau

```
request = new ActiveXObject('Microsoft.XMLHTTP');
```

Un pic de istorie: Internet Explorer permitea extinderea funcționalității browser-ului prin intermediul unor așa numite controale ActiveX, de fapt un fel de plugin-uri. Toate "viewer"-urile integrate în Internet Explorer care permiteau vizualizarea diferitelor formate grafice în acest browser (precum fișiere pdf, animați Flash, applet-uri Java) erau oferite de fapt sub forma de plugin-uri (controale) ActiveX. Pe primele versiuni de Internet Explorer (\leq IE6) inclusiv funcționalitatea Ajax era oferită sub forma unui control ActiveX.

Exemplu discutat, disponibil online [aici](#): Se cere scrierea unui formular de înregistrare a unui utilizator în cadrul unei aplicații web. Formularul va conține câmpurile uzuale prezente într-un astfel de formular: username, adresa de e-mail și parola de două ori. Înainte de a se face submit la formular și a trimite toate datele despre noul utilizator la back-end, să se verifice printr-un apel AJAX unicitatea existenței numelui de utilizator pe back-end (utilizatorul nu va putea face submit dacă numele de utilizator este deja folosit).

Se recomandă rularea acestui exemplu (și a tuturor exemplelor din prezentul material cu Developer Tools pornit pe tab-ul de Network). De asemenea, vă rog să consultați codul sursă al acestor exemple! Alte funcționalități ale prezentei probleme (precum compararea celor două parole pe front-end, submit-ul efectiv al formularului) nu au fost implementate pentru a păstra codul relevant pentru partea de AJAX cât mai concis.

Front-end: fișier index.html

```
<script type="text/javascript">
```

```
var cancontinue = false;  
// variabila declarata în scopul global (dată membră pe window) pentru a putea fi  
// accesată de peste tot. Dacă este true se poate face submit la formular
```

```
function verifica(usernameInput) {  
    // usernameInput - inputul unde se introduce username-ul
```

```
    var request;  
    var username = usernameInput.value; // valoarea din input  
    var statusImg = document.getElementById('statusImg');  
    // imaginea care animeaza starea apelului AJAX
```

```
    request = new XMLHttpRequest(); // creăm apelul
```

```

// funcția anonimă de mai jos nu se execută acum! Acum se execută doar o atribuire
request.onreadystatechange = function() {
    // la momentul execuției funcției, dacă apelul AJAX s-a termina și e OK (200)
    if (request.readyState == 4)
        if (request.status == 200)
            if (request.responseText == 1) {
                // dacă de pe back-end soșeste în urma apelului AJAX un 1
                // username-ul este disponibil
                statusImg.src = 'ok.png'; // bifă verde
                cancontinue = true;
                // setăm variabila din scopul global la true, putem face submit la form
            }
            else { // altfel, username-ul este folosit deja
                statusImg.src = 'deny.png';
                cancontinue = false;
                // setăm variabila din scopul global la false,
                // nu putem face submit la form
            }
        }
    statusImg.src = 'loading.gif';
    // Doar un gif animat care simbolizează apelul AJAX în desfășurare

    request.open('POST', 'verif.cgi', true);
    request.send('username=' + username);
    // Facem call-ul AJAX efectiv trimițând prin POST numele de utilizator
}

</script>
</head>
<body>
    <form method="post" action="#" onsubmit="return cancontinue;">
        Nume utilizator: <input type="text" name="username" id="username"
onblur="verifica(this)">
        <!-- Evenimentul onblur se apelează când inputul pierde focusul -->
        &nbsp;  <br>
        E-mail: <input type="text" name="email"><br>
        Parola: <input type="password" name="pass"><br>
        Parola din nou: <input type="password" name="pass2"><br>
        <input type="Submit" value="Register">
    </form>
</body>
</html>

```

Back-end: verific

Back-end-ul nu este important în momentul de față. Îl prezentăm totuși, cu o scurtă explicație. Este vorba de un fișier .cgi care extrage username-ul primit prin POST în momentul în care se desfășoară call-ul AJAX, și caută acest username într-un fișier în care sunt memorati utilizatorii înregistrați deja („baza de date” cu utilizatori). Dacă numele de utilizator s-a regăsit printre cei înregistrați, întoarce spre front-end un „0”, altfel întoarce spre front-end un „1”.

```

#!/bin/bash
echo Content-type: text/html
echo
sleep 2 # simulam o cautare intr-o baza de date cu milioane de utilizatori :)
read dataFromClient
user=`echo $dataFromClient | cut -d"=" -f2`
if grep $user useri.dat > /dev/null
then

```



```
echo 0
else
echo 1
fi
```

Observație importantă din punct de vedere al securității: Orice verificări care se fac pe front-end se fac doar “de dragul” de a face interfața cu utilizatorul cât mai prietenoasă. Verificările de pe front-end trebuie dublate de verificări pe back-end care sunt absolut vitale din punct de vedere al securității.

Spre exemplu se pot compara cele două parole pe front-end în exemplu de față pentru a nu lăsa utilizatorul să continue înregistrarea și să facă submit la formular dacă parolele sunt diferite. Însă această verificare trebuie ulterior făcută și pe back-end în scriptul care preia toate datele din formular și face înregistrarea efectivă a utilizatorului în baza de date. De asemenea, verificarea unicității username-ului trebuie făcută din nou pe back-end la momentul efectiv al inserării acestuia în baza de date, din simplu motiv că de la completarea numelui de utilizator în inputul corespunzător și până la submiterea formularului se poate scurge o perioadă de timp în care altcineva se poate înregistra cu username-ul respectiv. Validările pe back-end trebuie făcute în primul rând pentru că cele de pe front-end nu sunt sigure, utilizatorul putând face disable la execuția codului JavaScript sau poate altera codul JavaScript care se execută în browser folosind Developer Tools.

Un exemplu mai complex

Problema rezolvată [aici](#): Într-o tabelă a unei baze de date memorate pe back-end sunt stocate trenuri, fiecare tren fiind caracterizat de oraș plecare, oraș sosire, oră, minut. Folosind două componente de tip `select` și apeluri AJAX să se afișeze sub forma unui tabel orarul acestor trenuri.

Codul este un pic cam lung pentru al prezenta integral în documentul de față, dar dăm în continuare câteva explicații pentru rezolvarea părții de front-end (back-end-ul nu ne interesează în acest moment), explicații care pot fi urmărite pe codul sursă al exemplului de față.

`Select`-ul ce conține orașele de plecare este precompletat pe back-end (cod server side) odată cu generarea documentului HTML ce se trimite clientului. La schimbarea valorii selectate, în `select`-ul `plecare` (`onchange` pe aceste element) se execută funcția `getArrivals()`. Aceasta funcție va iniția un apel AJAX (pe care puteți să-l urmăriți în tabul Network din Developer Tools) care va returna de pe back-end o expresie JSON ce va conține toate localitățile de sosire în care se poate ajunge din localitatea de plecare selectată. O expresie JSON (abreviere de la JavaScript Object Notation) poate fi evaluată ușor la un obiect (`array` în cazul de față) JavaScript folosind `eval` (cam periculos...) sau `JSON.parse()`. Acest `array` va conține localitățile unde se poate ajunge din localitatea de plecare selectată și va fi folosit pentru a popula `select`-ul cu `id`-ul `plecare`.

Observații:

În exemplu de față apelul AJAX întoarce datele (stațiile de sosire) încapsulate sub forma unei expresii JSON. Am specificat anterior, că, în funcție de tehnologia care se folosește pe back-end și front-end se poate alege și o altă formă de încapsulare a datelor, spre exemplu XML (de unde și numele AJAX).

Spunem în cazul de față că apelul AJAX se face spre un end-point (care returnează un `array` de localități). Ne putem gândi la un end-point ca la un anumit script care implementează și executa o anumită logică pe back-end și care este apelat prin protocolul HTTP.

Apelul AJAX făcut în cadrul funcției `getArrivals` este făcut prin GET. În foarte multe exemple disponibile online apelurile AJAX sunt făcute prin POST, sau prin alte metode HTTP, cum ar fi PUT, DELETE, PATCH când apelurile se fac spre servicii web REST (dar despre asta în posibil alt curs, probabil la MPP ☺). În unele situații este mai ușor de efectuat apelul AJAX prin GET, un apel prin GET fiind mai ușor de depanat (mai ușor de făcut „debugging”). Spre exemplu, se poate încărca manual în browser (cerere care se face de fapt prin GET) un URL de forma:

<https://www.scs.ubbcluj.ro/~bufny/pw/ajax/trenuri/cautaDestinatii.php?plecare=iasi>

pentru a vedea dacă „end-point-ul” întoarce corect pentru localitatea de plecare Iasi o expresie JSON ce conține localitățile unde se poate ajunge din Iasi. Dacă acesta se comporta corect, poate fi apelat și prin intermediul unui call AJAX. O depanare similară dacă end-point-ul ar fi fost apelat prin POST nu ar fi fost posibilă.

Odată completat dinamic `select`-ul `sosire` pe baza răspunsului primit în urma primului apel AJAX, se poate face un nou call AJAX (click-ul pe „Afișează trenurile” apelează funcția `getTrains()`) spre cel de-al doilea end-point care întoarce, tot sub forma unei expresii JSON, toate trenurile din baza de date între localitățile selectate. Spre exemplu, pentru `plecare=iasi` și `sosire=Timisoara` (URL complet end-point <https://www.scs.ubbcluj.ro/~bufny/pw/ajax/trenuri/cautaTrenuri.php?plecare=iasi&sosire=Timisoara>) se obține următoarea expresie JSON:

```
{
  "trenuri": [
    { "plecare": "Iasi", "sosire": "Timisoara", "ora": 10, "minut": 45},
    { "plecare": "Iasi", "sosire": "Timisoara", "ora": 12, "minut": 30},
  ]
}
```

Aceasta expresie este din nou convertită la un `array` JavaScript ce este iterat pentru a popula sub forma unui tabel containerul `mersulTrenurilor`.

Apeluri AJAX din jQuery

Librăriile și framework-urile JavaScript pot oferi în general modalități mai simple și mai elegante de a efectua apeluri AJAX. Astfel, în jQuery sunt prezente următoarele funcții care permit toate efectuarea de astfel de apeluri:

- `$.ajax`
- `$.get`
- `$.post`
- `load`

Exemplu de mai jos (disponibil online [aici](#)), reia primul exemplu din acest material ce cerea back-end-ului convertirea unui text la majuscule (back-end-ul este același pentru ambele exemple):

```
<script type="text/javascript" src="jquery.min.js"></script>
<script type="text/javascript">
```

```
$(function() {
  $("#sir").keyup(function () {
```

```

$.ajax({
    type: 'GET',
    url: 'toUpper.php',
    data: 'sir=' + $("#sir").val(),
    success: function(majuscule){
        $("#rezultat").html(majuscule);
    }
});
});
});

```

</script>

Introduceti un sir: <input type="text" id="sir">

 Sirul trimis la server si primit inapoi convertit la majuscule este:

\$.get și \$.post permit și ele realizarea de apel-uri AJAX însă sunt mai puțin customizabile decât \$.ajax. Cea mai puțin customizabilă/parametrizabilă modalitate de a face apel AJAX din jQuery este folosind load, dar pe de altă parte aceasta este și cea mai simplă. load populează un container cu output-ul obținut în urma unui apel AJAX prin GET către un end-point. Folosind load, exemplu de mai sus poate fi rescris astfel (disponibil online [aici](#)):

```

<script type="text/javascript" src="jquery.min.js"></script>
<script type="text/javascript">

```

```

$(function() {
    $("#sir").keyup(function () {
        $("#rezultat").load(encodeURIComponent("toUpper.php?sir=" + $("#sir").val()));
    });
});

```

</script>

Introduceti un sir: <input type="text" id="sir">

 Sirul trimis la server si primit inapoi convertit la majuscule este:

Funcția encodeURIComponent a fost folosită în exemplu de mai sus pentru că se pare ca load nu acceptă URL-uri ce conțin spații, un end-point de forma toUpper.php?sir=ana are mere nefiind apelat corect.

Alte exemple

Am spus mai sus că funcția specificată ca și valoarea pentru proprietatea onreadystatechange se apelează când obiectul AJAX își schimbă starea (valoarea proprietății readyState) de la 0 la 4. Mai există un caz, interesant, în care se poate apela această metodă: atunci când apelul AJAX este în starea 3 (adică sunt date în curs de primire de la back-end dar acestea nu s-au terminat) și valoarea proprietății.responseText se modifică (tot „crește” pe măsură ce sosesc noi date). Pentru a ilustra acest comportament, prezentăm următorul exemplu, disponibil online [aici](#) (a se vizualiza cu Developer Tools pornit în tab-ul Console).

Front-end, fișier index.html:

```

<script type="text/javascript">

```

```
function go() {
    var request = new XMLHttpRequest();

    console.log('Ready state: ' + request.readyState + ' | Response text: ' +
request.responseText);
    request.onreadystatechange = function() {
        console.log('Ready state: ' + request.readyState + ' | Response text: ' +
request.responseText);
    }

    request.open('GET', 'go.cgi', true);
    request.send('');
}

</script>
<input type="button" value="Go!" onclick="go()">
```

Back-end, fișier go.c, sursa fișierului go.cgi:

```
#include <stdio.h>

int main() {
    setvbuf(stdout, NULL, _IONBF, 0);
    printf("Content-type: text/html\n\n");
    int i ;
    for (i = 0; i < 10; i++) {
        printf("%d ", i);
        fflush(stdout);
        // necesar pentru a goli stream-ul spre client (browser)
        sleep(2);
        // simulam printr-un sleep o chestie care dureaza mult pe back-end
    }
    return 0;
}
```

În exemplu de mai sus, se face un singur apel AJAX (poate fi vizualizat în tab-ul Network din Developer Tools), dar funcția de call-back pentru `onreadystatechange` se apelează de mai multe ori când apelul AJAX este în starea 3.

Let's play: on-line Minesweeper

[Click here](#) for a new game!

Observații:

- Mai are câteva bug-uri :D
- Pe client (browser) nu se știe unde sunt bombele – acestea sunt păstrate pe back-end. Back-end-ul e scris în PHP, detalii în cursul următor...
- Pentru a vizualiza codul sursa a front-end-ului în Firefox și Chrome precedați cu “view-source:” URL-ul joculețului.

Sunt deschis la orice sugestii de îmbunătățire a acestui material și observații privind eventuale scăpări / greșeli (acord bonusuri recompensă ☺). Mulțumesc.