Bevetés közben – Ismerkedés az Ajaxszal

Hogy kerül az A – mint aszinkron – az Ajaxba?

Sok programozó, így én is, jó ideje ismeri a *Javascript*-et, mellyel dinamikusan módosíthat *HTML* oldalakat. Persze más apró feladatra is alkalmas, ilyen például az űrlap érvényességének ellenőrzése. Az utóbbi években azonban a *Javascript* lett a húzóerő az alkalmazásfejlesztők körében, köszönhetően az *Ajax*-os megoldásoknak.
A *Javascript* népszerűsége előtt egy-az-egyben megfeleltetés volt a felhasználó reakciója és a *HTML* oldal megjelenítése között. Ha ráklikkelt a felhasználó egy linkre, akkor az aktuális oldal eltűnt és betöltődött az új. Ha elküldött egy űrlapot, akkor azt megkapta a webkiszolgáló és a választ jelenítette meg. A hagyományos webalkalmazásoknál szerveroldalon dolgozták fel a felhasználói adatokat és a szerver oldalon rakták össze a dinamikus oldalakat is.
Az *Ajax*-os alkalmazások megosztják a terhelést azzal, hogy nagyobb szerepet kap a kliens oldali *Javascript*. Számos *Ajax* program teljes *HTML* oldalt generál le, amely aztán egy-az-egyben jelenik meg a webböngészőben. A többinél azonban a szerver csupán apró *XML* formátumú morzsákat ad a kliensnek. Ezt a kliens kéri és dolgozza fel *Javascript*-tel, majd pedig frissíti a *HTML* oldal egy részét anélkül, hogy a teljes oldalt újra kellene tölteni vagy le kellene cserélni. A webes *DOM* (*document object model*) és *CSS* (*cascading stylesheets*) segítségével az *Ajax*-os alkalmazások ugyanazokkal a tulajdonságokkal bírnak – használhatóság, felhasználóbarát, azonnali válasz –, mint azt az asztali alkalmazásoknál megszokta a felhasználó.
Az elmúlt pár hónap után most folytatjuk a kliens oldali *Javascript* és *Ajax* felfedezését. Az előző hónap témája a felhasználók webes regisztrációja volt. Noha a tényleges regisztráció szerver oldali, olyan megoldás kerestünk, ami *Ajax* segítségével figyelmezteti a felhasználót, ha az adott azonosító már foglalt. Természetesen szerver oldali ellenőrzést is használhatnánk, ez azonban az oldal frissítésével jár, ami időbe telik.
A múlt hónapban bemutatott megoldás a felhasználó számára megfelelő (különösen, ha kedveli a spártaian egyszerű kinézetet). Azonban a problémát nem igazán *Ajax*-os módon oldottuk meg. Egy tömbben beégetve tároltuk a felhasználói neveket és ebben a tömbben kerestünk. A megvalósítás számos sebből vérzik, hiszen bárki megtekintheti a már regisztrált felhasználók azonosítóit, illetve sok felhasználó esetén a tömb óriásira dagad, így sokáig tart letölteni az oldalt. Az oldal letöltési ideje és az abban való keresés időtartama a regisztrált felhasználók számának növekedésével arányosan nő.
Ezek a problémák *Ajax*-os megoldással elkerülhetők. A felhasználói lista *Javascript* forrásba drótozása és a teljes lista lekérése helyett csupán megkérdezzük a szervertől, hogy az adott felhasználó létezik-e már? Ez elég gyors letöltést és reakcióidőt eredményez amellett, hogy áttekinthetőséget és egyszerű bővíthetőséget ad.
Most belevetjük magunkat az *Ajax*-ba. A korábbi szerver és kliens oldali programot módosítjuk úgy, hogy aszinkron módon kérje le a felhasználóneveket. Eközben látni fogjuk, milyen egyszerű *Ajax* alkalmazást készíteni, vagy egy meglévő web alkalmazást felruházni *Ajax*-os funkciókkal. A cikk végére a kedves Olvasó is képes lesz hasonló szerver- és kliens oldali *Ajax* alkalmazásokat írni.

Ajax hívás

A *Javascript*-es *XMLHttpRequest* objektum teszi lehetővé az *Ajax*-os lehetőségek nagy részét. Ennek az objektumnak a segítségével indíthat *HTTP* kérést egy *Javascript* függvény és ennek segítségével reagálhat rá. (Biztonsági okok miatt az *XMLHttpRequest* objektum csak azzal a szerverrel tud adatot cserélni, amelyről letöltődött az adott oldal.) A *HTTP* kérés egyaránt lehet *GET* vagy *POST*, azonban az utóbbival tetszőleges hosszúságú és bonyolultságú kéréseket indíthatunk.

A legérdekesebb és egyben az ***Ajax*** paradigma alapja, hogy az ***XMLHttpRequest*** indíthat ***szinkron*** (a böngészőnek várnia kell, amíg a teljes válasz megérkezik) és ***aszinkron*** (a böngésző használható, miközben érkeznek az adatok) ***HTTP*** kéréseket is. Az ***Ajax*** rendszerint aszinkron hívásokkal dolgozik. Ez lehetővé teszi, hogy a weboldal egyes részeit egymástól függetlenül frissítsük, voltaképpen egy időben válaszol többszörös adatbevitelre.
Elméletileg az alábbi ***Javascript*** sorral hozhatunk létre egy példányt az ***XMLHttpRequest*** objektumból:

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

Sajnos az élet nem ilyen egyszerű. Azért nem, mert a legtöbb ember ***Internet Explorer***t használ. Az ***Explorer*** nem rendelkezik beépített ***XMLHttpRequest*** objektummal, így nem is példányosítható az említett módon. Így azonban igen:

```
var xhr = new ActiveXObject
➥("Msxml2.XMLHTTP");
```

Álljunk csak meg! Néhány ***Explorer*** verziónál ez azonban kicsit másképp néz ki:

```
var xhr = new ActiveXObject
➥("Microsoft.XMLHTTP");
```

Hogyan kezeljük le ezt a három különböző ***XMLHttpObject*** példányosítást? Az egyik megoldás: detektáljuk szerver oldalon a böngészőt. Persze ugyanez kliensoldalon is megoldható. De a legelegánsabb módra, amivel mostanáig találkoztam – ***Michael Mahemoff: Ajax Design Patterns*** című könyvében leltem. ***Mahemoff*** a ***Javascript*** kivételkezelését használja, amíg valamelyik nem lesz jó. A három megoldást egy függvénybe helyezve és ezt rendelve az `xhr` változóhoz, biztosíthatjuk alkalmazásunk keresztplatfromos működését:

```
function getXMLHttpRequest () {
try { return new ActiveXObject
➥("Msxml2.XMLHTTP"); }
➥catch(e) {};
try { return new ActiveXObject
➥("Microsoft.XMLHTTP"); }
➥catch(e) {}
try { return new XML
➥HttpRequest(); } catch(e) {};
    return null;
}
var xhr = getXMLHttpRequest();
```

A fenti részlet futtatásakor az `xhr` értéke vagy `null` (ez azt jelenti, hogy nem tudta példányosítani az `XMLHttpRequest` objektumot) vagy pedig az `XMLHttpRequest` objektum az adott böngészőben megfelelő megvalósítása. A példányosítás után már van egy keresztplatformos `XMLHttpRequest` objektumunk, így továbbá nem kell figyelnünk rá.
A leggyakoribb eljárás az `open`. Ez inicializálja a ***HTTP*** kérést egy bizonyos ***URL***-re a forrás szerver felé. Jelen esetben például így hívhatjuk meg az `xhr` példányunk `open` eljárását:

```
xhr.open("GET", "foo.html",
➥true);
```

Az első paraméter (***GET***) megmondja az `xhr.open`-nek, hogy ***GET*** típusú ***HTTP*** lekérdezés lesz. A második paraméter megadja az ***URL***-t, amit szeretnénk elérni. Jegyezzük meg, minthogy a forrás kiszolgálójához csatlakozunk, a protokoll és a szerver címe hiányzik. A harmadik paraméterrel megadjuk, hogy ***aszinkron*** (`true`) vagy ***szinkron*** (`false`) kérést szeretnénk-e indítani? Csaknem minden ***Ajax*** alkalmazásban `true` az értéke, ami azt jelenti, hogy a böngészőben megnyitott oldal addig is használható, amíg a ***HTTP*** kérés válaszát várja.
Az aszinkron ***HTTP*** az ***Ajax*** fő vonzereje. Minthogy a ***HTTP*** kérés nem befolyásolja a felhasználói felületet, így a webes alkalmazás sokkal inkább helyi, asztali alkalmazásnak tűnik.
Az `xhr.open()` még nem jelent ***HTTP*** kérést, csupán beállítja az objektumot, a küldéskor az itt beállított paramétereket fogja használni. A kérés elküldésére ezt használjuk:

```
xhr.send(null);
```

Az `XMLHttpRequest` soha nem ad vissza ***HTTP*** választ, akárki hívja is meg a `xhr.send()` eljárást. Ennek oka, hogy az `XMLHttpRequest`-et aszinkron módban használjuk, hiszen az `xhr.open()`-nél true-ra állítottuk ezt a paramétert. Nem tudjuk megjósolni, hogy fél másodpercen, öt másodpercen, 1 percen vagy 10 órán belül kapunk választ. Ehelyett, megmondjuk a ***JavaScript-nek***, hogy mely függvényt hívja meg, ha megérkezik a válasz. Ez a függvény fogja beolvasni és feldolgozni a választ, illetve ez hajtja végre a szükséges utasításokat.
A `parseHttpResponse` egyszerű megvalósítása így néz ki:

```
function parseHttpResponse() {
    alert("entered parseHttp
➥Response");
    if (xhr.readyState == 4) {
        alert("readystate ==
➥4");
        if (xhr.status == 200) {
            alert
➥(xhr.responseText);
        }
        else
        {
            alert("xhr.status
➥== " + xhr.status);
        }
    }
}
```

A `parseHttpResponse` meghívásra kerül, ha megérkezik a válasz az ***Ajax***-os kérésünkre. Hogy biztosak legyünk, megérkezik-e a teljes válasz, figyeljük a `xhr.readyState` attribútumát. Ha ez ***4***, akkor kapott az `xhr` választ. Következő lépésben ellenőrizzük a válasz ***HTTP*** kódját. A sikeres ***(OK)*** lekérésnek a ***kódja 200***. Természetesen kaphatunk ***404-et ("file missing")*** a szervertől, de az se kizárt, hogy egyáltalán nem tudtunk kommunikálni a szerverrel. Ahhoz, hogy a ***JavaScript*** meghívja a `parseHttpResponse` függvényünket ***HTTP*** válasz esetén, állítsuk át az `XMLHttpRequest` objektum `onreadystatechange` attribútumát:

```
xhr.onreadystatechange =
➥parseHttpResponse;
```

Végül, miután megbizonyosodtunk róla, hogy megkaptuk a választ és minden rendben, a szöveget a `xhr.responseText` eljárással nyerhetjük ki. Az ***XMLHttpRequest***-től kétféle formátumú adatot kaphatunk: egyszerű szöveg (mint itt is) vagy ***XML*** dokumentum. Utóbbi esetben használhatjuk a ***DOM***-ot navigációhoz, akárcsak egy weboldal esetén.

1. Lista ajax-test.html

```
<!DOCTYPE html PUBLIC “-//W3C//DTD XHTML 1.0 Strict//EN”
“http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd”>
<html xmlns=”http://www.w3.org/1999/xhtml”>
  <head><title>Ajax test</title>
    <script type=”text/javascript”>
    function getXMLHttpRequest () {
    try { return new ActiveXObject(“Msxml2.XMLHTTP”); }
    ➥catch(e) {};
    try { return new ActiveXObject(“Microsoft.XMLHTTP”); }
    ➥catch(e) {}
    try { return new XMLHttpRequest(); } catch(e) {};
        return null;
    }
    function parseHttpResponse() {
        alert(“entered parseHttpResponse”);
        if (xhr.readyState == 4) {
            alert(“readystate == 4”);
            if (xhr.status == 200) {
                alert(xhr.responseText);
            }
            else
            {
                alert(“xhr.status == “ + xhr.status);
            }
        }
    }
    var xhr = getXMLHttpRequest();
    alert(“xhr = “ + xhr);
    xhr.open(“GET”, “atf.html”, true);
    xhr.onreadystatechange = parseHttpResponse;
    xhr.send(null);
    </script>
  </head>
  <body>
    <h2>Headline</h2>
    <p>Paragraph</p>
  </body>
</html>
```

Természetesen egy *Ajax* alkalmazás nem hívja meg minden lépésben az `alert` függvényt. Helyette valami hasznosabban csinál: szöveget módosít vagy a dokumentumfához ad ágakat vagy töröl belőle, de akár a kinézetet is módosíthatja. A forráskód az *1. Listában* olvasható.
Noha az ***ajax-test.html*** egyszerű, mégis egy teljes értékű *Ajax* program. A kipróbáláshoz szükségünk van a webszerveren a `DocumentRoot` könyvtárban az ***atf.html*** állományra. (Különben ***404-es HTTP*** hibakódot kapunk.) Ha felmerült a kérdés, vajon mennyire lehet bonyolult egy *Ajax* hívás, akkor a példa mutatja, hogy viszonylag egyszerű.

Ajax-os regisztráció

Most, hogy láttuk, hogy működik egy *Ajax*-os program, a tudás birtokában módosítsuk a múlt havi regisztrációs programunkat. A korábbi megoldásnál a *JavaScript*-ben definiáltuk a felhasználói azonosítókat. Ha a felhasználó egy olyan azonosítót kért, ami már jelen volt a rendszerben, úgy jelezte azt és nem engedte a regisztrációt. Nem írom le az összes problémát ezzel a megközelítéssel kapcsolatban, mert sok volt. Egyszerű alternatívaként mi lenne, ha *Ajax*-al érnénk el a felhasználói azonosítók listáját? Ebben az esetben biztosak lehetünk, hogy naprakész a lista.
Mi lenne, ha a fix, előre bedrótozott lista helyett a webszerverről kérnénk le azt? (Természetesen ez nem olyan kulturált megoldás, mintha igent vagy nemet kapnánk egy bizonyos felhasználói névre. Arról a következő hónapban beszélek.) Ha az *Ajax*-os lista dinamikusan generálódna, akkor a szükséges adatokat adatbázisból is kinyerhetnénk. Ezt *XML*-be elküldve egyszerűen betölthető lenne a tömbbe. Hogy a mostani példánkon egyszerűsítsek, statikus oldalt használunk dinamikus helyett. Természetesen ha a kedves Olvasó korábban már írt szerveroldali webalkalmazásokat, úgy semmiség a statikus fájlt dinamikussá alakítani.
A regisztrációs oldal, amely ezt a listát értelmezi, alább található. Az ***ajax-register.html*** fájl hasonló a múlt havihoz. A nem *Ajax*-os megoldásnál tömbben (`usernames`) tároltuk a felhasználói azonosítókat. Definiáltuk a `checkUsername` függvényt, melyet a `username` szöveges mező `onchange` kezelőjéhez rendeltünk hozzá.

2. Lista usernames.txt

```
abc
def
ghi
jkl
mno
pqr
stu
vwx
yzz
```

A függvény így akkor hívódott meg, ha a felhasználó befejezte a kívánt felhasználói név begépelését. Ha az már regisztrálva volt, úgy a felhasználó kapott egy figyelmeztetést és a véglegesítő gomb inaktívvá vált. Különben a felhasználó elküldhette az adatot a szerveroldali alkalmazásnak, jelezve hogy szeretne regisztrálni.
Hogy a múlt havi regisztrációs oldalt *Ajaxossá* alakítsuk, módosítjuk a `checkUsername` függvényt. Ez akkor hívódik meg, ha a felhasználó befejezte a felhasználói név bevitelét.

3. Lista ajax-register.html

```
<!DOCTYPE html PUBLIC “-//W3C//DTD XHTML 1.0
➥Strict//EN”
“http://www.w3.org/TR/xhtml1/DTD/xhtml1-
➥strict.dtd”>
<html xmlns=”http://www.w3.org/1999/xhtml”>
  <head><title>Register</title>
    <script type=”text/javascript”>
    function getXMLHttpRequest () {
        try { return new ActiveXObject
➥(“Msxml2.XMLHTTP”); } catch(e) {};
        try { return new ActiveXObject
➥(“Microsoft.XMLHTTP”); } catch(e) {};
        try { return new XMLHttpRequest(); }
➥catch(e) {};
        return null;
    }
    function removeText(node) {
        if (node != null)
        {
        if (node.childNodes)
        {
            for (var i=0 ; i <
➥node.childNodes.length ; i++)
            {
            var oldTextNode = node.childNodes[i];
            if (oldTextNode.nodeValue != null)
            {
                node.removeChild(oldTextNode);
            }
            }
        }
        }
    }
        function appendText(node, text) {
            var newTextNode =
➥document.createTextNode(text);
            node.appendChild(newTextNode);
        }
        function setText(node, text) {
            removeText(node);
            appendText(node, text);
        }
        var xhr = getXMLHttpRequest();
        function parseUsernames() {
            // Set up empty array of usernames
            var usernames = [ ];
            // Wait for the HTTP response
        if (xhr.readyState == 4) {
        if (xhr.status == 200) {
            usernames =
➥xhr.responseText.split(“\n”);
        }
        else
        {
            alert(“problem: xhr.status = “ +
➥xhr.status);
        }
        }
            // Get the username that the person
➥wants
            var new_username =
➥document.forms[0].username.value;
            var found = false;
            var warning = document.getElementById
➥(“warning”);
            var submit_button =
➥document.getElementById(“submit-button”);
        // Is this new username already taken?
Iterate over
        // the list of usernames to be sure.
            for (i=0 ; i<usernames.length; i++)
            {
                if (usernames[i] == new_username)
                {
                    found = true;
                }
            }
        // If we find the username, issue
➥a warning and stop
        // the user from submitting the form.
            if (found)
            {
                setText(warning, “Warning:
➥username ‘“ + new_username
                +”’ was taken!”);
                submit_button.disabled = true;
            }
            else
            {
                removeText(warning);
                submit_button.disabled = false;
            }
        }
        function checkUsername() {
            // Send the HTTP request
        xhr.open(“GET”, “usernames.txt”, true);
        xhr.onreadystatechange = parseUsernames;
        xhr.send(null);
        }
    </script>
  </head>
  <body>
    <h2>Register</h2>
    <p id=”warning”></p>
    <form action=”/cgi-bin/register.pl”
➥method=”post”>
        <p>Username: <input type=”text”
➥name=”username”
onchange=”checkUsername()” /></p>
        <p>Password: <input type=”password”
➥name=”password” /></p>
        <p>E-mail address: <input type=”text”
➥name=”email_address” /></p>
        <p><input type=”submit” value=”Register”
➥id=”submit-button”/></p>
    </form>
  </body>
</html>
```

Az előredefiniált tömb helyett a `checkUsername` *Ajax*-os kérést indít a szerver felé. A múlt havi nem *Ajax*-os változathoz képest most csupán ennyit csinál a `checkUsername`. A frissített függvény így néz ki:

```
function checkUsername() {
xhr.open("GET",
➥"usernames.txt", true);
xhr.onreadystatechange = parse
➥Usernames;
xhr.send(null);
}
```

Ahogy látható, lekéri a *usernames.txt*-t a szerverről. Ha az `xhr` állapota megváltozik, meghívásra került a `parseUsernames` függvény. A függvényt komoly dolgokkal vérteztük fel. Először is a kapott állományt tömbbé alakítjuk:

```
var usernames = [ ];
if (xhr.readyState == 4) {
if (xhr.status == 200) {
    usernames =
➥xhr.responseText.split("\n");
}
}
```

Itt ismét belebotlunk a korábbi *Ajax*-os példába: várunk, amíg az `xhr.readyState` értéke *4* lesz, majd leellenőrizzük a `xhr.status`-t (a *HTTP* válaszkódja) hogy *200*-e? Itt már tudjuk, hogy hibátlanul megkaptuk a *usernames.txt* fájl tartalmát. Ez tartalmazza a már regisztrált felhasználók listáját. Egy felhasználói név egy sorban, amint az a 2. Listában is látszik. A *JavaScript* `split` függvényét használva a `usernames` tömbbe helyezzük a felhasználói neveket.
Innentől már használhatjuk a múlt havi nem *Ajax*-os megoldást. Először is a *DOM* segítségével lekérdezzük pár elem azonosítóját:

```
var new_username =
➥document.forms[0].username.
➥value;
var found = false;
var warning = document.
➥getElementById("warning");
var submit_button = document.
➥getElementById
➥("submit-button");
```

Ezután megnézzük, hogy az éppen begépelt felhasználói név szerepel-e a tömbünkben:

```
for (i=0 ; i<usernames.length;
➥i++)
{
    if (usernames[i] ==
➥new_username)
    {
        found = true;
    }
}
```

Ha szerepel, akkor figyelmeztető üzenetet írunk az oldal tetejére. Különben töröljük az esetleges figyelmeztetéseket:

```
if (found)
{
    setText(warning, "Warning:
➥username '" + new_username
➥+"' was taken!");
    submit_button.disabled =
➥true;
}
else
{
    removeText(warning);
    submit_button.disabled =
➥false;
}
}
```

Nézzük csak, megfelelő módon kezeljük a felhasználói neveket? Nem igazán, hiszen most még csak nagyon kezdetleges módon implementáltuk az *Ajax*-ot. A hatékonyságon és a biztonságon javíthatunk.
Az egyik probléma a statikus fájl. Természetesen a szerveren a *cron* segítségével időnként legeneráltathatnánk a *usernames.txt* állományt, ez azonban eléggé fapados megoldás. Helyette használhatunk szerveroldali programot adatbázis lekérdezéssel megtámogatva. Statikus oldalról dinamikusra váltani már csak a teljesítmény javítása miatt is jó ötlet.
Biztonsági okok is vannak. A múlt havi programunkkal a teljes felhasználói lista is letöltésre került. Ez azt jelenti, hogy rosszhiszemű felhasználó betekintést kap a felhasználók listájába. Ez lehetővé teszi, hogy betörjön az oldalára vagy kéretlen üzenetekkel halmozza el.
Az ilyen *Ajax*-os ellenőrzés egyik hátulütője a sebesség. Ahogy azt már korábban jeleztem, az *Ajax* aszinkron mivoltából adódik, sose tudható, mennyi időn belül kapunk választ. Az én esetemben a böngésző és a szerver közötti adatforgalomra szinte nem kellett várni. Egy terhelt szerver, egy összetettebb adatbázis lekérdezés vagy lassú internet kapcsolat esetén azonban már lomhának érezhetjük az aszinkron hívásokat. Azonban még a legrosszabb *Ajax* függvény is gyorsabb, mint a teljes oldal újratöltése, hiszen kevesebb az adatátvitel.

Zárszó

Ebben a részben végre elkezdtük használni az *Ajax*-ot az alkalmazásukban. Láttuk, miként kell egy meglévő *JavaScript* programot két függvényre bontani: az egyik az *Ajax* hívásért felel, a másik pedig feldolgozza a választ.
Természetesen láttuk a megoldás biztonsági és hatékonysági korlátait is. Jobb megoldás elküldeni csupán a felhasználói nevet és egy egyszerű igen/nem választ várni a szervertől, hogy foglalt-e már a felhasználói név. A következő hónapban a mostani *GET* helyett *POST* kérést fogunk küldeni és a statikus *usernames.txt* lecseréljük szerveroldali alkalmazásra, amely az *Ajax* hívásunkkal fog együttműködni.

Ajánlott olvasmányok

Az utóbbi időszakban robbanásszerűen bővült az *Ajax*-os irodalom, nehezen tudok velük lépést tartani. Két nagyon jó könyv van a témában. Az egyik a *Head Rush Ajax*. Ez első sorban a kezdőket célozza meg és hatékony, szórakoztató módon vezet be a rejtelmekbe. A másik a már korábban említett *Ajax Design Patterns*, amely minden bizonnyal a kedvenc *Ajax*-os könyvem (a kinézete és a felépítése ellenére, amely nem követi a szokásos *O'Reilly* hagyományokat). Ezutóbbi bevezetőnek ajánlott a gyakorlott webfejlesztőknek.
Az *Ajaxian.com* weboldal rengeteg linket, tananyagot és cikket tartalmaz *Ajax* fejlesztés témában többféle platformra. Érdemes felvenni az oldalt a *Kedvencek* közé vagy az *RSS* olvasónkba.

Linux Journal 2006., 151. szám

Reuven M. Lerner