Szabad navigáció okozta problémák webes környezetben

Hogyan kezeljük helyesen a böngésző Vissza/Tovább, és Frissítés műveleteinek használatakor, hivatkozásokra többszöri kattintáskor, direkt navigációkor és gyorsítótár használatakor felmerülő problémákat webes alkalmazások fejlesztésekor

Viczián, István

Ez a cikk azon problémával foglalkozik, mely a legtöbb webes alkalmazás fejlesztésekor felmerül, ugyanis nem biztosítható az, hogy a felhasználó olyan sorrendben nézze meg az oldalakat, ahogy azt az alkalmazás fejlesztője eltervezi. Használhatja a Vissza és Tovább műveleteket is navigációra, valamint újratöltheti az oldalt a Frissítés művelettel. Ezen műveletek elérhetők a böngésző szokásos gombjai között, billentyűkombinációval, de jobb kattintásra felugró menüben is. Sokan megszokásból, esetleg türelmetlenség (, a lassú válaszidő) miatt duplán kattintanak egy adott hivatkozásra. A felhasználó kézzel is beírhat egy url-t, vagy a Kedvencek közül is választhat egyet, ami szintén hibás működéshez vezethet, ha erre nem készülünk fel, és bízunk, hogy csak a felületi elemeket (űrlap elemek – gomb, legördülő menü, stb., hivatkozások) fogja használni. A böngészők és tűzfalak gyorsítótár beállításai is megzavarhatják az előre tervezett munkafolyamatot. A probléma a webes technológia, a http(s) protokoll, valamint a böngészők adta lehetőségek miatt jelentkezik.

Vegyük észre, hogy nem egy pontosan megfogalmazható problémáról van szó, hanem egy probléma csoportról. Ez a csoport a következő problémákat tartalmazza, melyeket ezek után összefoglaló néven szabad navigáció okozta problémáknak nevezünk:

- A Vissza és Tovább navigációs műveletek szabad használata
- A Frissítés művelet használata
- Többszörös kattintás (pl. hosszabb válaszidő esetén)
- Közvetlen URL használata vagy a Kedvencek közül, vagy bármilyen más forrásból

Egy bizonyos alkalmazásnál lehet, hogy a fenti problémáknak csak részhalmaza jelentkezik, és valószínű, hogy alkalmazásonként máshogy kell kezelnünk. Függhet ez az alkalmazás architektúrájától, használt technológiáktól, a felhasználók képzettségétől és szokásaiktól, illetve, hogy maga az alkalmazás milyen funkciókkal rendelkezik (, és azok lefutási idejétől). Ebből talán látható, hogy egy általános megoldás nem adható, mely az összes problémát megszünteti, viszont vannak elterjedt technikák, melyek megfelelő kiválasztásával és implementálásával az alkalmazásunk felkészíthető az ilyen esetek kezelésére. Ez a cikk ebben próbál segítséget nyújtani, és széles körben, de még mindig nem eléggé, elterjedt megoldási javaslatokat adni.

A cikk gyakran fog említeni tervezési mintákat és újratervezési megoldásokat. Természetesen a szorosan kapcsolódó mintákat és megoldásokat kifejtem a cikkben, de a lazán kapcsolódó fogalmaknak az

olvasónak kell utánanéznie, ahogy az alkalmazott technológiáknak, keretrendszereknek is. Ehhez a cikk végén található irodalomjegyzék is segítséget nyújt.

1. Környezet

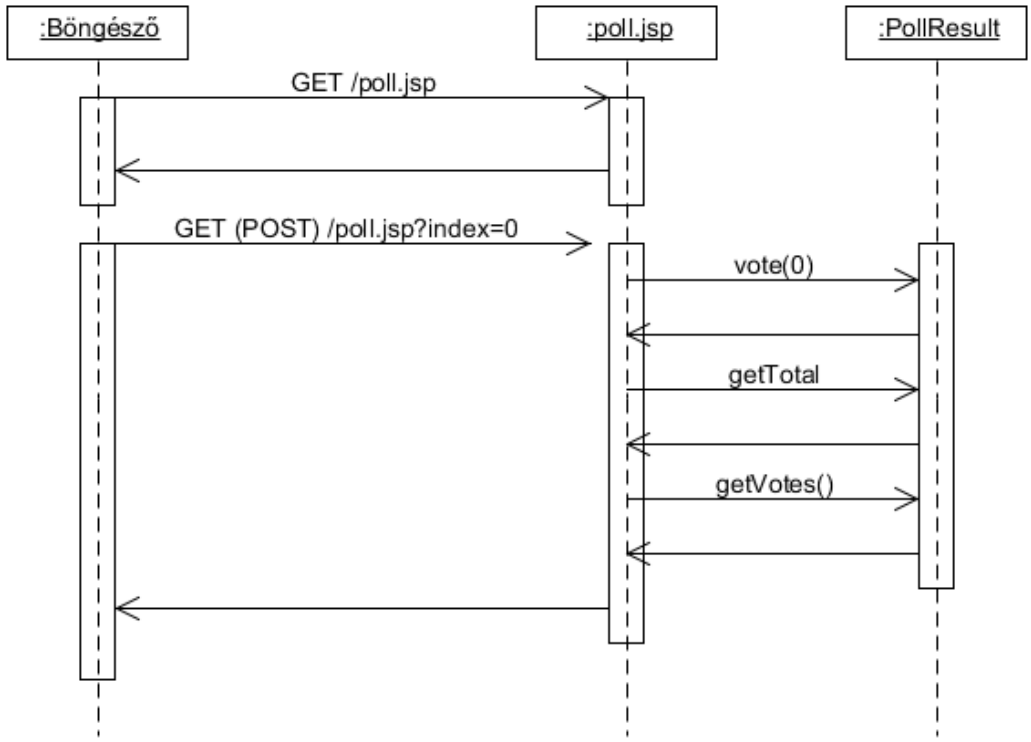
A cikk alapvetően két fajta megközelítést fog tárgyalni, egyrészt a kliens oldali, másrészt a szerver oldali megoldásokat. A cikkhez tartozó példaprogram elérhető a http://https://github.com/vicziani/jtech-log-repost címen, innen akár egy zip állományban is letölthető. A példaprogram több példát is tartalmaz, melyek elérhetők a főoldalról. A cikk szövegében a példákra azok számával fogok hivatkozni. Ez a szám megtalálható az állományok és csomagok neveiben is. A kliens oldali megoldások nagy része JavaScript nyelven implementálható. Szerver oldali megoldások példáit Java nyelven közlöm, felhasználva a Servlet 3.0/JSP 2.2 specifikációkat is. Ezek tetszőlegesen átírhatók más keretrendszerre (Struts, Spring MVC, Spring WebFlow, Tapestry, Wicket, JSF) vagy technológiát használva (pl. ASP, PHP, Perl, Python, stb.), sőt a hivatkozások között ilyen megoldások is megtalálhatóak. A megvalósítás során a Java SE Development Kit 6 fejlesztőkörnyezetet, valamint Jetty web-konténert használtam, a letölthető példák is e környezeten futtathatóak. A projektek Maven 3-mal fordíthatóak, valamint minden konfiguráció nélkül futtathatóak a projekt főkönyvtárában az **`mvn jetty:run`** parancs kiadásával (elvégzi a fordítást és összecsomagolást is). Ekkor a konzolon a `Started Jetty Server` üzenet jelenik meg, és az alkalmazás elérhető a http://localhost:8080/ címen. A folyószövegben az Internet Explorer által használt fogalmakat fogom használni (Internet Explorer 8), de ki fogok térni a Firefox böngésző terminológiájára is (Firefox 7.0.1).

2. Példa

A hibajelenségek leírásához és megoldási javaslatok prezentálásához egy sokak által ismert, egyszerű példát fogok használni, ami egy szavazatszámláló alkalmazás. Nem cél egy teljes megoldás megvalósítása, csupán a hibajelenségek prezentálása és megoldási lehetőségek bemutatása. Ezért a példa nem tartalmaz teljeskörű megvalósítást a szálbiztosságra, szavazatok tartós tárolására (állomány, adatbázis), valamint arra, hogy egy felhasználó csak egyszer szavazhasson. Ez utóbbi esetben felhasználó-kezelést kellene megvalósítani, melynek sokfélesége túlmutat ezen cikk keretein.

Először egy meglehetősen hibás programból indulok ki (0. példa), hogy az összes hibalehetőséget prezentálni lehessen rajta. Az alkalmazás egy `poll0.jsp` JSP lapból és egy `PollResult` bean-ből áll.

A JSP lap attól függően, hogy az index névvel szerepel-e paraméter a kapott paraméterek között, kétféleképpen viselkedik. Ha az oldal nem kapott paramétert, kitesz három rádiógombot valamint egy Elküld gombot. Az Elküld gombra kattintva GET http metódussal elküldésre kerül az `index` paraméter (jelezve, hogy melyik rádiógomb lett kiválasztva), és ezt érzékelve a JSP meghívja a bean `vote` metódusát, amely elvégzi a megfelelő számláló növelését, majd vár 1 másodpercet. Eztán megjeleníti a szavazás eredményét, lekérdezve az értékeket a bean-től (a választási lehetőségre leadott szavazatok, valamint az összes szavazat száma). Az egy másodperces várakozás azért van be építve, hogy az Elküld gombon nyomott többszörös kattintást tesztelni lehessen.




```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  <link rel="stylesheet" href="css/jtechlog.css" type="text/css"/>
        <title>Szavazás</title>
    </head>
    <body>
        <c:choose>
            <c:when test="${empty param.index}">
                <form method="get">
                    <p>Melyiket részesíted előnyben?</p>
                    <ul>
                        <li><input type="radio" name="index" value="0"
      checked="true" />Java</li>
                        <li><input type="radio" name="index" value="1" />.NET</li>
                        <li><input type="radio" name="index" value="2" />Egyéb</li>
                    </ul>
```

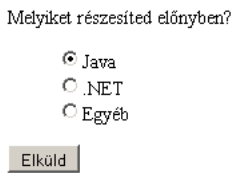
```
                    <p>
                        <input type="submit" name="Submit" value="Elküld" />
                    </p>
                </form>

            </c:when>
            <c:otherwise>
                <jsp:useBean id="result" scope="application"
                             class="jtechlog.repost.PollResult" />
                <%
                    result.vote(Integer.parseInt(request.getParameter("index")));
                %>
                <p>Eddigi szavazatok: ${result.total}</p>
                <ul>
                    <li>Java: ${result.votes[0]}</li>
                    <li>.NET: ${result.votes[1]}</li>
                    <li>Egyéb: ${result.votes[2]}</li>
                </ul>
            </c:otherwise>
        </c:choose>
    </body>
</html>
```

A bean tartalmaz egy `votes` nevű tömb mezőt a lehetőségenkénti szavazatok számlálására, és egy `total` nevű mezőt az összes szavazat számlálására.

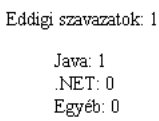
```
package jtechlog.repost;

/**
 * Szavazatokat gyűjti.
 */
public class PollResult {

    private int votes[] = new int[3];

    private int total = 0;

    /**
     * Szavazás.
     *
     * @param index válasz indexe
     */
    public synchronized void vote(int index) {
        total++;
        votes[index]++;
        try {
            // Amiatt iktatunk be várakozást, hogy a dupla kattintást
            // tesztelni lehessen
            Thread.sleep(1000);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }

    /**
```

```
     * Visszaadja a leadott szavazatok számát válaszonként.
     */
    public int[] getVotes() {
        return votes;
    }

    /**
     * Visszaadja az összes szavazat számát.
     */
    public int getTotal() {
        return total;
    }
}
```

Mikor betöltjük az oldalt a `pollapp/poll.jsp` címen, akkor először az űrlap jelenik meg.



Kiválasztva egy lehetőséget, és az Elküld gombot megnyomva a böngésző GET (vagy POST) metódussal elküldi a szervernek a kiválasztott rádiógombhoz rendelt értéket. Mivel ekkor már van paraméter, a JSP lap meghívja a bean vote metódusát, ami vár is egy másodpercet, majd a JSP kiírja a szavazás eredményét.



Az alkalmazással használatakor a következő problémák merülhetnek fel:

1. A felhasználó a Vissza műveletet használva visszanavigálhat az első oldalra, és újra leadhatja a szavazatát.

2. Amennyiben a metódus get, a második oldalon egy Frissítés gombot nyomva a szavazat újra leadásra kerül.

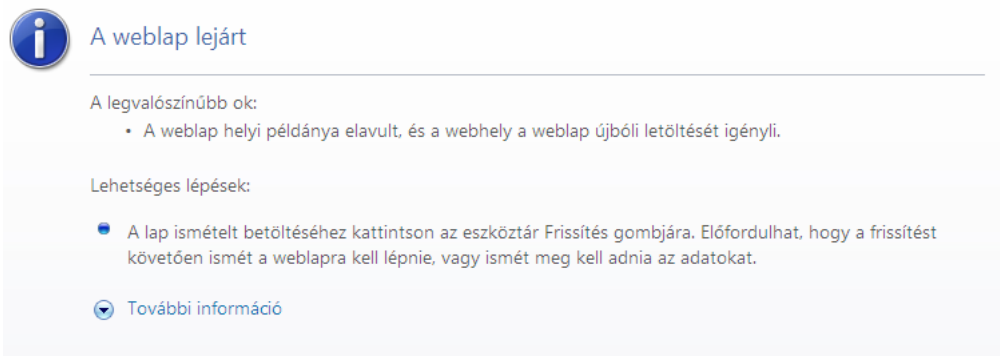
3. A metódust írjuk át get-ről post-ra a `form` tag `method` paraméterében, és úgy lépjünk vissza az első oldalra, frissítsük azt, majd ismét nyomjuk meg az Elküld gombot, majd azon az eredmény oldalon a Frissítés gombot (1. példa). Egy figyelmeztető oldal jelenik meg, mely arra kéri a felhasználót, hogy az oldal frissítéséhez nyomja meg az Ismét gombot. Ekkor a felhasználó választásához tartozó érték ismét nőni fog eggyel. Ezzel kapcsolatos, ha az eredmény oldal gyorsítótárazását letilt-

juk, majd leadjuk a szavazatunkat, és a Vissza, majd az Előre gombot nyomjuk meg, a böngésző figyelmeztetni fog, hogy a weblap lejárt (kizárólag az Internet Explorer - 2. példa). A Firefox nem ad ilyen üzenetet ([Deabill2008]).

A gyorsítótár letiltására a JSP 5. sorába illesszük a következő kódrészletet:

```
<%-- Cache-elés letiltása. --%>
<%
response.setHeader("Cache-Control","no-cache"); // HTTP 1.1
response.setHeader("Pragma","no-cache"); // HTTP 1.0
response.setDateHeader ("Expires", -1); // Proxy servernek jelzi a cache tiltását
%>
```

Az Internet Explorer a következő üzenetet jeleníti meg:



4. A hosszabb várakozási idő miatt (amit most mesterségesen generálunk, de független lehet az alkalmazástól, okozhatja pl. hálózati hiba, adatbázis lassulás is) a felhasználó többször is kattinthat az Elküld gombra, ami azt eredményezi, hogy a választásához tartozó számláló akár többet is ugorhat.

5. Ha a felhasználó a második oldalt elmenti a Kedvencek közé, vagy kimásolja, és utána újra behívja, újra szavazást fog leadni.

A következő táblázat mutatja a különböző műveletek elérhetőségét.

1. táblázat - Műveletek különböző böngészőkben

Művelet	Eszköztárban (IE)	Eszköztárban (Firefox)	Felugró menüben (IE és Firefox)	Billentyűzet kombináció (IE és Firefox)
Vissza	Vissza gomb az eszköztárban	Ugrás az előző oldalra	Jobb kattintás/ Vissza	**Alt + Balra nyíl**
Előre	Előre gomb az eszköztárban	Ugrás a következő oldalra	Jobb kattintás /Előre	**Alt + Jobbra nyíl**

Művelet	Eszköztárban (IE)	Eszköztárban (Firefox)	Felugró menüben (IE és Firefox)	Billentyűzet kombináció (IE és Firefox)
Frissítés	Frissítés gomb az eszköztárban	Aktuális oldal újratöltése	Jobb kattintás/Frissítés	**F5**

Amennyiben az Internet Explorer-ben a frissítést a **Ctrl + Frissítés**, vagy a **Ctrl + F5** gomb lenyomásával végezzük, a böngésző nem veszi figyelembe a cache-t. Firefox esetén ugyanez elérhető a **Ctrl + Shift + R**, **Ctrl + F5** kombinációkkal, vagy **Shift + Aktuális oldal újratöltése** gomb kombinációval.

Ezekkel a problémákkal gyakran lehet találkozni a weben, a következő kulcsszavakkal lehet rákeresni: disable back button, duplicate clicking, multiple submits, duplicate submissions, synchronizer token, dejavu token, deja vu token, vagy ezek kombinációja. Az irodalomjegyzékben több ilyen cikk is megtalálható.

3. Többszöri kattintás és JavaScript megoldások

Az többszöri kattintás probléma kiküszöbölésére több kliens oldali megoldás is akad, bár egyik sem ad tökéletes megoldást, kritikusabb alkalmazásoknál mindenképp meg kell támogatni szerver oldalon is. Hiszen minden böngészőnél kikapcsolható a JavaScript-ek futtatása, illetve a cookie-k használata, valamint lehetnek olyan kliensek, melyek eleve nem támogatják ezeket a lehetőségeket.

Először mindenképpen meg kell vizsgálni az alkalmazást sebesség szempontjából, hiszen ez a hiba csak akkor merülhet fel, ha a válasz viszonylag lassan érkezik a felhasználó böngészőjéhez. Ilyenkor ugyanis a felhasználó megállíthatja a letöltést, és újra elküldheti az űrlapot (esetleg egyből a Frissítés műveletet használva). Ilyenkor a szerveren ugyanahhoz a felhasználóhoz több kiszolgáló szál is elindul, és az első szálak eredményét ugyan előállítja a szerver, de nem használja fel, hiszen nem adhatja vissza a böngészőnek, hiszen az már új kérést indított. Ekkor fokozatosan nő a szerver terhelése, a felhasználók egyre türelmetlenebbek lehetnek, és többször próbálják a műveletet elvégezni, ami végső esetben a szerver összeomlását is okozhatja.

A többszöri kattintás megakadályozása azáltal, hogy gyorsítunk az alkalmazáson, főleg az intranetes hálózaton elhelyezett, egyidejűleg kevés felhasználó által használt, kis terheltségű, gyors szerver esetén használható, hiszen pl. internetes alkalmazás esetén a hálózati kommunikáció lassúsága (akár egy hálózati eszköz hibája) miatt is jelentkezhet a hiba.

Meg kell vizsgálni ezen kívül, hogy mennyire tér el egymástól az, ha egy felhasználó kétszer kattint gyorsan egymás után, illetve több felhasználó kattint egyszerre külön gépen. A második esetben mindenképpen működnie kell az alkalmazásnak minden különösebb felkészítés nélkül is. Egyszerű lekérdezéseknél az első esetben is hiba nélkül kell működnie, probléma a módosításoknál, illetve a lassú lekérdezések esetén lehetséges. Ha a probléma sebesség problémára is visszavezethető, érdemes elgondolkodni gyorsítótár alkalmazásán is.

Legegyszerűbb megoldásként érdemes figyelmeztetni a felhasználót arra, hogy ne nyomja meg a gombot semmi esetben sem kétszer, hiszen ez hibás működéshez vezethet. Persze ez nem véd, sőt a felhasználónak felhívjuk a figyelmét, hogy a hiba kihasználható.

Hatékonyabb megoldás az "Elküld" gomb letiltása annak megnyomása esetén. Erre a legegyszerűbb megoldás a gomb `onclick` tulajdonságának a használata (3. példa):

```
<input type="submit" name="Submit" value="Elküld"
 onclick="this.disabled=true; this.form.submit();"/>
```

A Firefox esetén amennyiben a Vissza művelettel visszatérünk az első lapra, a Elküld gomb még mindig letiltott állapotban van, és a Frissítés műveletre sem lesz újra aktív. Esetleg az űrlap `onsubmit` tulajdonságára is tehető hasonló funkcionalitás.

Persze ennél szebb megoldás is elkészíthető, pl. a gomb CSS visiblity property-jének állítása, esetleg még egy üzenet kiírása, miszerint a kérés feldolgozása folyamatban van. Sőt akár a teljes űrlapot el lehet tüntetni. Ezen megvalósításoknak csak a JavaScript/DHTML/CSS tudásunk szabhat határt, amelyek az AJAX térhódításával amúgy is rendkívül divatos technológiák. Ezen kívül valamikor érdemes egy ésszerű időn belül újra visszaállítani a gomb megnyomhatóságát, ha pl. kommunikációs hiba miatt megszakadt az átvitel, akkor újra lehessen kezdeni.

Másik kliens oldali megoldás az lehet ([Thomason2002]), hogy a böngésző cookie-ban tárolja el, hogy az adott űrlapot a felhasználó már elküldte (4. példa). Ekkor az újabb gombnyomásra a felhasználót erre figyelmeztetni lehet. A következő JavaScript-et az űrlap onsubmit tulajdonságában lehet meghívni.

```
function disableResubmit() {
    if (document.cookie.indexOf("voted") > -1) {
        alert(" Már szavaztál! "); return false;
    }
    else {
        document.cookie = "voted"; return true;
    }
}
```

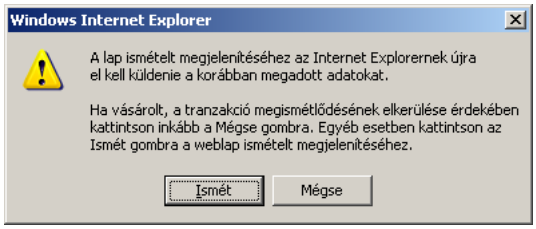
És a hozzá tartozó űrlap:

```
<script type="text/javascript" src="js/poll.js" charset="UTF-8"></script>
...
<form action="poll.jsp" method="post" onsubmit="return disableResubmit();">
```

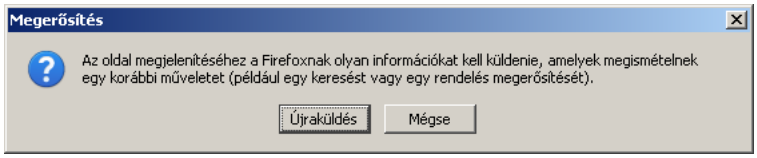
Ez a probléma hasonló ahhoz, mint mikor a felhasználó kivárja a válasz oldal letöltődését, majd visszanavigál az első képernyőre, és ismét megnyomja az Elküld gombot. Így a többszörös kattintásra adott megoldások kiküszöbölik ezt a problémát is.

4. Frissítés művelet és Redirect After Post

A Frissítés műveletre a böngésző feltesz egy kérdést, melyre igennel válaszolva újraküldi az űrlapot a szerver felé, ezért nő ismét a szavazatok száma. E kérdés megjelenítése helyes működés, mely a get és post metódusok közti eltérésből adódik. Az Internet Explorer a következő képernyőt jeleníti meg.



A Firefox a következő képernyőt jeleníti meg.



Ahogy a RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1, azaz a HTTP/1.1 specifikációja is írja, a get metódus való arra, hogy a szerverről bizonyos adatokat kérjünk le, és a post metódus való arra, hogy a szervernek adatokat küldjünk, és ezen küldés hatása lehet a szerveren tárolt adatok megváltozása. A get metódus nem változtathat meg semmilyen adatot a szerveren, csak lekérésre használható, ezért "biztonságos metódusnak" ("safe method") nevezzük. Másrészt a get metódus "idempotens", mely azt jelenti, hogy egynél több kérés mellékhatásának azonosnak kell lennie egy kérés mellékhatásával. A post metódusról ez nem mondható el, hiszen adatot módosít a szerveren, így két kérés eredménye akár teljesen más lehet.

Természetesen itt üzleti adatok módosításáról van szó, különböző egyéb módosulások történhetnek a szerveren, ilyen pl. a napló állomány írása (pl. access log). Fontos, hogy ezeket a szabályok betartását a protokoll maga nem biztosítja, hanem nekünk kell ezt betartani alkalmazásunk fejlesztésekor. Így a kérdés, melyet a böngésző feltesz, teljesen jogos, hiszen a "Frissítés" gomb hatására a szerveren újabb adatmódosítás történik, ezért mindenképp meg kell ezt erősíteni.

4.1. Böngésző oldali gyorsítótárazás

Interaktív webes alkalmazásoknál sok helyen nem használható a böngésző oldali gyorsítótárazás. A különböző grafikákat, CSS állományokat, statikus tartalmakat persze gyorsítótárazhatja a böngésző. Kellő körültekintéssel bizonyos dinamikus oldalak, vagy azok részei is gyorsítótárazást, mely az alkalmazásunk több szintjén is megjelenhet egészen a perzisztens szinttől a JSP darab szintig, de akár kliens oldalon is, ahol az adatok ritkán vagy előre becsült időközönként változnak, ilyenek pl. az online katalógusok.

Korábban láthattuk, hogy mi történik akkor, ha a cache-elést letiltjuk, és mégis a navigáció használatával újra az adott oldalra kerülünk. Mivel az előbb említett szabály szerint a post metódus üzleti adatok módosítására szolgál, a böngésző nem jelenítheti meg azt újra, hiszen a felhasználó beleegyezése nélkül nem küldheti el újra a szervernek a kérés paramétereit, hiszen az érzékeny adatok módosulásával járna.

Az üzenet, a külön oldal zavaró, nem illeszkedik be az alkalmazás megjelenésébe, és túl technikai, így csak megzavarhatja a felhasználót. Ez a jobb eset, van olyan böngésző, mely kérdés nélkül újraküldi az oldalt, mely beláthatatlan következményekhez vezethet. A Firefox böngésző figyelmen kívül hagyja ezeket a beállításokat, mindenképpen cache-el, de legalább nem küldi el újra.

A get metódusú kérésekre adott válaszban is letilthatjuk a gyorsítótárazást, ilyenkor nem jelenik meg a fenti üzenet, egyszerűen újra elküldi a kérést a szervernek. Ha az 5. példában (mely a get metódust használja) a Vissza/Előre navigációt használjuk, láthatjuk, hogy nem jelenik meg az üzenet, helyette viszont nőni fog a szavazatok száma.

A böngészők, proxyk és szerverek ilyen irányú vezérlésének hatékonyságáról erősen megoszlanak a vélemények.

Abban az esetben, ha szeretnénk, hogy a frissítés hatására az eredmény lista újra letöltődjön (hiszen pl. kíváncsiak vagyunk arra, hogy időközben szavaztak-e mások), de a szerveren adatmódosítás ne történjen; illetve a navigáció gombok miatti lap elévülést meg szeretnénk szüntetni, az alkalmazást két részre kell szétbontanunk. Egyrészt a post metódussal a jelenlegi működéshez hasonlóan adatot kell módosítani a szerveren, majd get metódussal le kell kérni az eredményt. Az utóbbi lapra akár közvetlenül is hivatkozhatunk, navigálhatunk, sőt a Kedvencek közé adhatjuk, lekérésekor nem kapunk hibaüzenetet, és nem módosítjuk a szavazás állását.

A problémát Michael Jouravlev nevesítette Double Submit néven, és a megoldásnak a Redirect After Post nevet adta, és a TheServerSide.com oldalon publikálta web alkalmazásokhoz használatos tervezési mintának (([Jouravlev2004])).

E megoldás használata során viszont eggyel több kérés-válasz utazik a böngésző kliens és a szerver között, így valamelyest lassítja az alkalmazást. Viszont ezt egyensúlyozza a HTTP specifikációban is leírt szabály betartása, az átláthatóbb alkalmazáslogika, valamint a helyes működés.

A két rész között egy átirányítást kell végezni. Ezzel egy tiszta Model – View – Controller megoldást kapunk. Első lépésként a post metódussal a vezérlő módosítja a modellt, majd második lépésként a modell alapján visszakapjuk a nézetet. A nézetre újabb kéréseket küldve nem változik a modell, azaz ha időközben más felhasználó nem módosítja az adatokat, ugyanazt az eredményt fogjuk kapni.

Ahhoz, hogy az MVC modellt teljessé tegyük, az alkalmazást Model 2 architektúrának megfelelően alakítjuk át.

4.2. Model 2

Kezdetben, mikor még csak a Servlet specifikáció létezett, minden munkát a servletek végezték, ők értelmezték a felküldött adatokat, módosították a modellt és jelenítették meg a tartalmat. Így a servlet forrásában HTML kódok szerepeltek, később sablonmotorokat használtak fel. Később jelent meg a JSP specifikáció, mely megfordította az irányt, méghozzá HTML kódba lehet Java kódot illeszteni (ez később finomodott a Tag Library-kkel és Expression Language-el). Ekkor jelentek meg a Model 1 JSP alkalmazások, mikor az adatok feldolgozását, átirányításokat és a megjelenítést is JSP lapok végezték. Mégis pont a felépítésük miatt hamar nyilvánvalóvá vált, hogy a servletek erőssége a bejövő adatok

feldolgozása, valamint a navigáció (átirányítások) biztosítása (MVC architektúrában a vezérlő), míg a JSP lapok erőssége a megjelenítés. Egy olyan alkalmazást, ahol a servletek végzik a vezérlő logikát, és JSP lapok a megjelenítést, Model 2-es alkalmazásnak hívunk.

Látszik, hogy a Model 2 és az MVC fogalom összefügg, de mégsem teljesen. Model 1 architektúrában is lehet MVC architektúrájú alkalmazást felépíteni, ebben az esetben csak arra kell figyelni, hogy a JSP lapokat két csoportra osszuk fel, egyik csoport, mely a vezérlést végzi, másik csoport, ami a megjelenítést. Az első csoport lapjai ne tartalmazzanak HTML kódot, és így tökéletesen meg tudnak felelni a szerepüknek, hiszen a JSP lapok is a web-konténerben servletként jelennek meg (előfordítva, vagy a konténer maga fordítja őket azzá).

Az alkalmazást tehát több részre bontjuk (6. példa). A klasszikus módszert alkalmazzuk, a vezérlő logikát servletek, a megjelenítést JSP oldalak végzik. Ez utóbbiakat innentől nem hívjuk közvetlenül, kizárólag servlet-en keresztül, így az `WEB-INF/jsp` könyvtárba kerülnek. A modell maradt a `PollResult` osztály, mely az átalakítás hatására sem változott. A `poll.jsp` egyszerűsödött, a feladata az űrlap megjelenítésére korlátozódott.

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />

        <link rel="stylesheet" href="<c:url value="/css/jtechlog.css" />"
   type="text/css"/>
        <title>Szavazás</title>
    </head>
    <body>
        <form method="post">
            <p>Melyiket részesíted előnyben?</p>
            <ul>
                <li><input type="radio" name="index" value="0"
     checked="true" />Java</li>
                <li><input type="radio" name="index" value="1" />.NET</li>
                <li><input type="radio" name="index" value="2" />Egyéb</li>
            </ul>
            <p>
                <input type="submit" name="Submit" value="Elküld" />
            </p>
        </form>
    </body>
</html>
```

Megjelent viszont egy új servlet `PollServlet` néven, mely feladata egyrészt az űrlap megjelenítése, valamint a válasz feldolgozása. Az űrlap megjelenítésekor egy forward történik a `poll.jsp`-re. A válasz

feldolgozásakor értelmezi a paramétert, elvégzi a szavazást, majd átirányít a megjelenítő rétegre, mely egy újabb JSP lap.

```
package jtechlog.repost.sample6;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import jtechlog.repost.PollResult;

/**
 * Az űrlapot jeleníti meg, valamint a
 * szavazást dolgozza fel.
 */
@WebServlet(urlPatterns = "/sample6/poll.html")
public class PollServlet extends HttpServlet {

    @Override
    public void doGet(HttpServletRequest req,
            HttpServletResponse res) throws ServletException, IOException {
        req.getServletContext().getRequestDispatcher("/WEB-INF/jsp/sample6/poll.jsp")
   .forward(req, res);
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
   throws ServletException, IOException {
        PollResult result = (PollResult) getServletContext().getAttribute("result");
        if (result == null) {
            result = new PollResult();
            getServletContext().setAttribute("result", result);
        }
        result.vote(Integer.parseInt(req.getParameter("index")));
        resp.sendRedirect("result.html");
    }
}
```

A szavazás állását egy külön servlet és JSP pár alkotja. A servlet ebben az esetben is csak egy forward-ot végez. A megjelenített lap akár fel is vehető a Kedvencek közé, sőt akárhányszor frissíthető, a modell nem fog ennek hatására változni, így ugyanazt jeleníti meg. A hibát jelző üzenetek sem fognak megjelenni a böngészőben.

```
package jtechlog.repost.sample6;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
/**
 * Szavazás eredményét jeleníti meg.
 */
@WebServlet(urlPatterns = "/sample6/result.html")
public class ResultServlet extends HttpServlet {

    @Override
    public void doGet(HttpServletRequest req,
            HttpServletResponse res) throws ServletException, IOException {
        req.getServletContext()
   .getRequestDispatcher("/WEB-INF/jsp/sample6/result.jsp")
   .forward(req, res);
    }
}
```

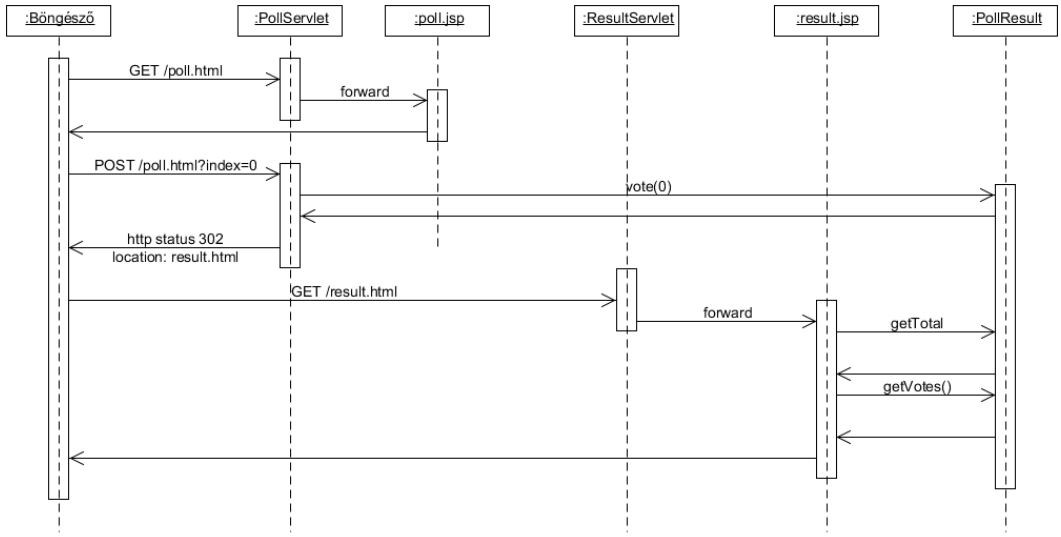
A `result.jsp` forráskódja a következő.

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<%
response.setHeader("Cache-Control","no-cache"); // HTTP 1.1
response.setHeader("Pragma","no-cache"); // HTTP 1.0
response.setDateHeader("Expires", -1); // Proxy servernek jelzi a cache tiltását
%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
 <link rel="stylesheet" href="<c:url value="/css/jtechlog.css" />" type="text/css"/>
        <title>Eredmény</title>
    </head>
    <body>
        <p>Eddigi szavazatok: ${result.total}</p>
        <ul>
            <li>Java: ${result.votes[0]}</li>
            <li>.NET: ${result.votes[1]}</li>
            <li>Egyéb: ${result.votes[2]}</li>
        </ul>
    </body>
</html>
```

A megoldás előnye, valamint a kiegészítő információk paraméterként való átadása mellett szóló érv, hogy ezzel a megoldással kihasználható a HTTP protokoll gyorsítótár szolgáltatása is (amit már jeleztünk, hogy csak indokolt esetben alkalmazzuk), míg a POST kérés eredménye sosem lesz gyorsítótárazva.

Az alkalmazás működése tehát a következő lesz:

- Űrlap lekérése (`poll.html`)
- A kérést a servlet dolgozza fel, és átirányítás történik a JSP-re
- Lefut a `poll.jsp`
- Az űrlapot a felhasználó visszakapja
- A felhasználó kiválaszt egy értéket, és felküldi az űrlapot
- A kontroller servlet a paramétert feldolgozza, és ennek megfelelően módosítja a modellt (a bean `vote` metódusát hívja), majd a böngészőnek egy átirányítást küld, átadva a megjelenítő komponens url-jét (`result.html`)
- A kérést a ResultServlet servlet dolgozza fel, majd forward történik a JSP lapra.
- A JSP lap lekéri a modellt az alkalmazás hatókörből, és kiolvasva annak értékeit legenerálja a megjelenítő lapot, és visszaküldi a felhasználónak



Az átirányításkor a háttérben tulajdonképpen ilyenkor az történik, hogy a `doPost` hívás egy rövid oldalt ad vissza 302-es (Found/Moved Temporarily) hibakóddal, valamint annak az oldalnak a címét a location nevű HTTP fejlécben, melyre a böngészőt továbbítani kell. Eztán a böngésző automatikusan lekéri az új oldalt a get metódussal.

Valójában erre az esetekre a HTTP 1.1 specifikációja a 303-as (See Other) hibakódot javasolja, és pont ezt a megoldási javaslatot adja a felmerült problémára. A 302-es és 303-as hibakódok közti egyik jelentős eltérés, hogy a 302-es hibakód hatására a böngésző nem változtathatja meg a kérés metódusát, tehát ha post volt az eredeti oldal lekérése, akkor az új oldalt is a post metódussal kell lekérni. Azonban a böngészőkben hibásan implementálták ezt a metódust, és úgy működnek, mintha a 303-as hibakódot

kapták volna, azaz a hivatkozott oldalt get metódussal kérik le. A későbbi problémák elkerülése céljából ajánlott a helyes 303-as hibakódot használni, így a `Response.sendRedirect` metódus a következő metódussal váltható ki (7. példa):

```
public void seeOther(HttpServletResponse response, String location)
  throws IOException {
    response.setHeader("Location", location);
    response.sendError(HttpServletResponse.SC_SEE_OTHER);
  }
```

Valószínűleg mivel ezt a hibát kihasználva jelentős mennyiségű alkalmazás készült, nem fogják a böngészők gyártói javítani, a specifikációhoz igazítani ezt a működést.

Sajnos a megoldás jellegéből adódóan ez az átirányítás nem helyettesíthető szerver oldali átirányításokkal (pl. `RequestDispatcher.forward`), hiszen akkor ugyanúgy egy darab POST metódusú kérés menne a szerver felé.

Ezen tervezési minta alkalmazása azonban nem azt jelenti, hogy egy űrlap nem használhat get metódust. Pl. egy adatok lekérésére szolgáló képernyőn, riport paraméterezés esetén, ahol a szűkítési feltételeket egy űrlapon lehet megadni, a metódus get legyen. A minta csak azt ajánlja, hogy a get metódusú kérés nem változtathat semmit az üzleti adatokon.

Ennek a technikának a használatakor jogos igény lehet, hogy a post után, annak feldolgozásakor kapott eredményeket adjuk át a következő, get metódussal lekérésre kerülő lapnak. Természetesen ez megoldható manuálisan is, vagy URL paraméterként adjuk át az adatokat, vagy a felhasználó munkamenetében, amit a get metódus kiszolgálásakor eltávolítunk.

Abban az esetben, ha az utolsó oldalnak még egyéb paramétert is át kell adni a (pl. a művelet eredményét), akkor ezt átadhatjuk url paraméterként vagy akár a munkamenetben is. Sajnos az átirányítás miatt egyszerűbb mód nem lehetséges.

A legelterjedtebb webes keretrendszerek beépített megoldást adnak erre, ahol a request, session és application scope mellett egy újabb hatókört vezettek be, keretrendszerenként más néven: Flash scope, Conversation Scope, Rollover Scope, Dialog Scope. Ennek az objektumnak az élettartama az aktuális kérés utáni kérésig tart. Természetesen az adat itt is a session-be kerül átadásra, csak az átirányított oldal kiszolgálása után kikerül onnan. A Struts 1-ben csak terv volt RolloverScope[1] néven, de nem valósították meg. Manuális megoldás, ha kézzel történik a session-ből az adat eltávolítása. Az Apache Tapestry `FlashPropertyPersistenceStrategy`[2] osztálya való erre, és dokumentációja szerint a Ruby on Rails-ből vették át. A Spring Web Flow `FlashScope`[3] osztályában jelenik meg. Sajnos sokáig a Spring MVC-ben nem létezett, kézzel kellett kivenni a session-ből a `SessionStatus`[4] osztály `setComplete()` metódusát hívva. A 3.1.0-ban viszont végre megjelent a Flash Scope[5].

[1] http://wiki.apache.org/struts/RolloverScope
[2] http://tapestry.apache.org/5.3/apidocs/org/apache/tapestry5/internal/services/FlashPersistentFieldStrategy.html
[3] http://static.springsource.org/spring-webflow/docs/2.3.x/javadoc-api/org/springframework/webflow/scope/FlashScope.html
[4] http://static.springsource.org/spring/docs/3.1.0.RC1/javadoc-api/org/springframework/web/bind/support/SessionStatus.html
[5] http://static.springsource.org/spring/docs/3.1.0.RC1/spring-framework-reference/html/mvc.html#mvc-ann-redirect-attributes

5. Vissza/Előre navigáció használata és a Synchronizer Token

A Vissza navigáció használatával a felhasználó visszamehet arra a lapra, ahol az űrlap található, és újra leadhatja szavazatát.

Ennek látszólagos kivédése lehet a Vissza navigáció működésének kliens oldali tiltása vagy megzavarása. Sajnos a webes technológia szabadsága miatt a felhasználóra nem kényszeríthetünk ilyen dolgokat, így egyik megoldás sem teljes, csupán felületes, és képzetlen felhasználók ellen nyújthat védelmet. Kizárólagos használatukat így nem javaslom.

A következő kliens oldali megoldások ismeretesek:

- Olyan ablak feldobása, mely nem tartalmazza a navigációs gombokat. Ez csak kezdő felhasználók ellen lehet jó megoldás, hiszen billentyűkombinációkkal, vagy a jobb kattintásra felugró menüből elérhetőek ugyanezen funkciók.

- A billentyűzet eseményeinek lekezelése, valamint a jobb kattintás letiltása. Böngészőnként különböző módon, de lehetséges JavaScript nyelven bizonyos események elkapása (pl. billentyű leütés, kattintás), és eseménykezelő implementálása, melyekkel a normál működés felülbírálható.

- Amennyiben egy oldalra, melyet nem akarunk közvetlenül a felhasználónak elérhetővé tenni, és ez linken keresztül érhető el, a klasszikus a tag helyett használhatjuk a `location.replace` JavaScript függvényt. Ekkor nem használható a Vissza gomb. Úgy képzelhető el, hogy nem hoz be egy új oldalt, hanem az következőt a létező helyére tölti be. Példa:

  ```
  <p><a href="JavaScript: location.replace('result.jsp');">Eredmény</a></p>
  ```

  Ez természetesen nem törli ki az űrlap előtt látogatott oldalakat a History-ból, így a Vissza gomb megnyomásakor a közbülső oldalt egyszerűen kihagyja.

- Másik egyszerű megoldás lehet, hogy arra az oldalra, ahova nem szeretnénk, hogy a felhasználó visszajusson (űrlap), elhelyezünk egy JavaScript-et, ami az oldal betöltődésekor indul el, és egyből a History következő elemére lép. Ekkor, amennyiben most került erre az oldalra, az megjelenik, hisz nincs következő oldal, ha a Vissza navigációval jut erre az oldalra, akkor a böngésző automatikusan továbbugrik arra az oldalra, ahonnan jött. A body onload eseményére tehető JavaScript:

  ```
  onload="if(history.length>0) history.go(+1);"
  ```

- Alternatív megoldás lehet, hogy az egymást követő oldalakat (űrlapokat) egy HTML oldalon helyezünk el, és JavaScript segítségével a különböző részeket hol eltüntetjük, hol megjelenítjük.

A JavaScript-es megoldások hátránya továbbá, hogy minden böngészőre fel kell készülni (bár ezt a modern JavaScript keretrendszerek kiküszöbölik), valamint a JavaScript kikapcsolható, sőt valamelyik kliens eleve nem is tudja futtatni azokat.

Ezen kívül a felhasználó nem csak a Vissza gomb megnyomásával juthat vissza egy előző oldalra, hanem egyszerűen az URL újbóli begépelésével, esetleg a Kedvencekből.

5.1. Synchronizer Token

Szerver oldalon a többszöri elküldést a felhasználó munkamenetének felhasználásával tudjuk megakadályozni. Erre több megoldás is létezik, ebből a Core J2EE Patterns könyvben ([CoreJ2eePatterns]) is megjelent un. Synchronizer Token alkalmazása. Ismeretes Déjà vu Token néven is, melynek alternatív írásmódjai is lehetégesek: dejavu, deja-vu, deja vu. A könyv szerzői csak megjelenítés rétegbeli tervezési megfontolások közé vették fel, és nem tervezési mintának, mivel azok egy magasabb szintű absztrakciós rétegbeli megoldások, és nem ennyire a technológiához kötöttek. Viszont a J2EE Refactoring fejezetben ismertetnek egy Introduce Synchronizer Token újratervezést.

Az alapötlet az, hogy felhasználónként egy egyedi azonosítót generálunk, és ezt eltároljuk a felhasználóhoz tartozó munkamenetben, illetve rejtett paraméterként az űrlapban, melyet ki kell töltenie. Az űrlap elküldésekor ellenőrizzük, hogy a munkamenetben tárolt és a paraméterként átadott token egyezik-e és sikeres feldolgozásakor kitöröljük a tokent a munkamenetből. Így abban az esetben, ha a felhasználó újra el akarná küldeni az oldalt, a token nem fog szerepelni a munkamenetben, de szerepelni fog a kérésben, így hibaüzenetet írunk ki.

A könyv a következő lépéseket ajánlja a Synchronizer Token bevezetésére:

- Írjunk egy vagy több segédosztályt (helper class), mely kezeli, generálja és összehasonlítja a tokeneket
- Írjuk meg az ellenőrzést, hogy a kérésben jött és a felhasználó munkamenetében tárolt token megegyezik-e
- Az újratervezési megoldás bevezetésekor érdemes alkalmazni az Introduce a Controller újratervezést is, ilyenkor egy helyen, centralizálva lehet megoldani a token kezelést. E nélkül minden egyes oldalban külön kell a token ellenőrzést elvégeznünk.

Az Apache Struts, mely egy nagyon elterjedt, nyílt forráskódú keretrendszer Java alapú webes alkalmazások fejlesztésére, beépítve tartalmaz egy megvalósítást a Synchronizer Token-re, emellett megvalósítja a Front Controller tervezési mintát is. Segédosztálya az `org.apache.struts.util.TokenProcessor`[6] osztály ([Reumann]). Ezt a megvalósítást (forráskódját), illetve használatát fogom itt is bemutatni (8. példa). A teljes forráskód a példa alkalmazásban megtalálható, itt a fontosabb részleteket közlöm.

```
public class TokenProcessor {

 // ...

    public synchronized boolean isTokenValid(HttpServletRequest request) {
        return this.isTokenValid(request, false);
    }
```

[6] http://struts.apache.org/api/org/apache/struts/util/TokenProcessor.html

```
public synchronized boolean isTokenValid(HttpServletRequest request,
    boolean reset) {
    // Retrieve the current session for this request
    HttpSession session = request.getSession(false);

    if (session == null) {
        return false;
    }

    // Retrieve the transaction token from this session, and
    // reset it if requested
    String saved =
        (String) session.getAttribute(Globals.TRANSACTION_TOKEN_KEY);

    if (saved == null) {
        return false;
    }

    if (reset) {
        this.resetToken(request);
    }

    // Retrieve the transaction token included in this request
    String token = request.getParameter(Globals.TOKEN_KEY);

    if (token == null) {
        return false;
    }

    return saved.equals(token);
}

public synchronized void resetToken(HttpServletRequest request) {
    HttpSession session = request.getSession(false);

    if (session == null) {
        return;
    }

    session.removeAttribute(Globals.TRANSACTION_TOKEN_KEY);
}

public synchronized void saveToken(HttpServletRequest request) {
    HttpSession session = request.getSession();
    String token = generateToken(request);

    if (token != null) {
        session.setAttribute(Globals.TRANSACTION_TOKEN_KEY, token);
    }
}

public synchronized String generateToken(HttpServletRequest request) {
    HttpSession session = request.getSession();
```

```
        return generateToken(session.getId());
    }

 // ...

}
```

A `TokenProcessor` a Singleton tervezési mintát követi, közvetlenül nem példányosítható, egyetlen példány létezhet belőle, és ezt a `getInstance` metódussal kell elkérni.

Az űrlapot megjelenítő `PollServlet doGet` metódusa hívja a token generálást, melyet a `org.apache.struts.util.TokenProcessor` osztály `saveToken` metódusa végzi. A token generálást a `generateToken` metódus végzi, mely veszi a munkamenet egyedi azonosítóját, valamint az aktuális rendszeridőt, ezeket bájttömbbé alakítja, majd egy MD5 checksum-ot generál hozzá, szintén bájttömb formátumban, és ennek adja vissza a hexadecimális reprezentációját (`toHex` metódus) egy `String`-ben. Ezután el kell helyezni az űrlapban egy rejtett mezőt, és értékül kell neki adni az immár munkamenetben tárolt tokent. Ehhez ismernünk kell, hogy a segédosztály milyen néven teszi be a tokent a munkamenetbe, valamint ellenőrzéskor milyen kérés paraméterként várja azt. Mindkettőt az `org.apache.struts.Globals` osztály tartalmazza, annak `TRANSACTION_TOKEN_KEY` és `TOKEN_KEY` konstanssa.

```
@Override
    public void doGet(HttpServletRequest req,
            HttpServletResponse res) throws ServletException, IOException {
        TokenProcessor.getInstance().saveToken(req);
        req.getServletContext().getRequestDispatcher("/WEB-INF/jsp/sample8/poll.jsp")
   .forward(req, res);
    }
```

Az űrlapva el kell helyeznie a tokent.

```
<input type="hidden" name="org.apache.struts.taglib.html.TOKEN"
 value="${sessionScope['org.apache.struts.action.TOKEN']}"/>
```

Az űrlapot feldolgozó servletnek az `isTokenValid(HttpServletRequest request, boolean reset)` metódust kell meghívnia. Ha igazzal tér vissza, akkor sikeres a művelet, ellenkező esetben kivételt dobunk. Hiba akkor következhet be, ha:

- Felhasználónak nem érvényes a munkamenete
- Ha a munkamenetben nem található a token
- Ha a kérésben nem található a token
- Ha a két token nem egyezik meg

A metódus második paramétere azt adja meg, hogy a tokent töröljük-e a munkamenetből. Ehhez a resetToken metódust hívja meg, amit közvetlenül mi is meghívhatunk.

A kivétel dobása elég drasztikus megoldás, megfelelő hibakezelést az olvasóra bízzuk. Például amennyiben a felhasználó duplaklikkelt, akkor hasznos lehet az a megvalósítás, mikor az első kérés eredményét egy rövidtávú cache-be helyezzük el, és a második kérést abból szolgáljuk ki.

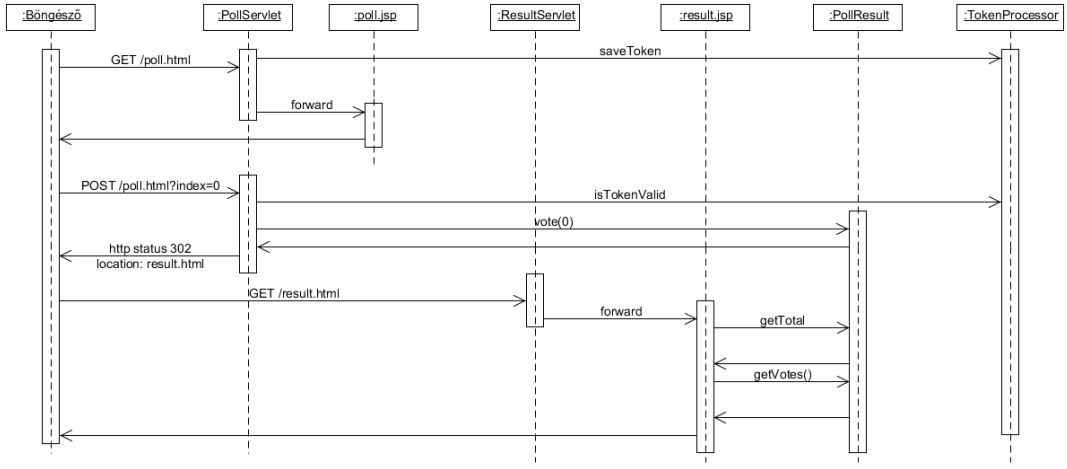
```
if (!TokenProcessor.getInstance().isTokenValid(req, true)) {
            throw new ServletException("Már szavazott!");
        }
```

A megoldás független attól, hogy a Frissít, Vissza/Tovább navigációs műveletek használata vagy többszörös kattintás miatt hajtódik végre kétszer a kérés.

Külön figyelnünk kell arra, hogy a tokent a feldolgozás elején vegyük ki a munkamenetből, ellenkező esetben ha a feldolgozás hosszabb ideig tart, akkor a második kérés még akkor érkezik be, mikor a token még nem lett eltávolítva. Legjobb, ha szinkronizálunk a munkamenet egyedi azonosítójára, így biztos, hogy ugyanazon blokk ugyanazon felhasználó bármennyi kérésére csak egyszer fog lefutni. A session objektum nem megfelelő erre a célra, hiszen a specifikáció nem biztosítja, hogy mindig ugyanazt a session objektumot kapjuk vissza, annak lekérésekor két kérés között (csak az biztosított, hogy a benne lévő objektumok lesznek ugyanazok).

Az alkalmazás a következőképpen működik:

- A felhasználó lekéri a /poll.html URL-en a szavazó űrlapot get metódussal.
- A kérés elmegy a `PollServlet` servlet `doGet` metódusához, mely generál egy tokent, és elhelyezi a munkamenetben, majd átirányít a `poll.jsp` JSP lapra (indirekt csak egy logikai nevet ad meg, melyet a struts-config.xml fordít fizikai névre)
- A `poll.jsp` visszaadja az űrlapot tartalmazó oldalt, és az űrlapba generálja a rejtett mezőt is, aminek értékül a munkamenetben található tokent adja
- A felhasználó kiválaszt egy rádiógombot, majd felküldi az űrlapot POST metódussal
- A kérés elmegy a `PollServlet` servlet `doGet` metódusához, mely ellenőrzi a tokent, ha az helyes, akkor elvégzi a szavazást, és innentől kezdve minden ugyanaz mint az előző példában. Amennyiben azonban a token nem valós, kivételt dob.



Egy másik ingyenes, nyílt forráskódú webes keretrendszer, az OpenSymphony WebWork is támogatja a Synchronizer Token tervezési mintát a `TokenHelper`, `TokenInterceptor`, `TokenSessionStore-Interceptor` osztályain keresztül. A The Java(TM) Developers Almanac 1.4, Volume 1 ([DevelopersAlmanac]) is tartalmaz egy egyszerű JSP-s megoldást, ahol csak egy timestamp-et tárol mind rejtett mezőben, mind a munkamenetben. Sajnos a Java ServerFaces specifikáció nem tartalmaz megvalósítást erre a tervezési mintára. A Spring MVC szintén nem tartalmaz erre beépített megoldást, azonban könnyen hozzá lehet fejleszteni ([Senior2009]). A Struts 2 a `Token`[7] osztálya lehet segítségünkre.

Abban az esetben, ha még jobban kontrollálni szeretnénk a felhasználó műveleteit, és szeretnénk, ha maximálisan betartaná az általunk megszabott sorrendiséget, akkor érdemes egy számlálót létrehozni (, és nem egy véletlenszerűen generált tokent), és azt hasonló módon tárolni a munkamenetben, és átadni a kérésekben. Ha űrlapról van szó, akkor az előző megoldáshoz hasonlóan egy rejtett mezőt alkalmazhatunk, de akár linkre is használhatjuk, és ekkor URL paraméterként kell átadni. Ebben az esetben azonban figyelnünk kell arra, hogy ez csak egy nyitott ablak esetén fog működni. Abban az esetben, ha a felhasználó, vagy esetleg egy JavaScript új ablakot nyit meg, akkor az ablakoknak egy egyedi azonosítót kell generálni, és minden ablakhoz nyilván kell tartani egy számlálót, hiszen a felhasználó a különböző ablakokat tetszőleges sorrendben használhatja. Az ablak azonosítóját minden kérésben tovább kell adni (ez a munkamenetben nem tárolható, hiszen a munkamenet felhasználóhoz tartozik, és nem ablakokhoz). Ekkor mivel ez még bonyolultabb megoldás, mint a Synchronized Token, mindenképp érdemes központilag, esetleg a front controller-ben megvalósítani.

Az előző megoldás ugyan kivédi a többszörös kattintás okozta problémákat, de ha visszanavigálunk az űrlapra, és újra lekérjük azt (ekkor frissül a token), majd felküldjük a szervernek az űrlapot, a kérést újra fel fogja dolgozni. Ennek kivédésére szintén lehet munkamenetet használni, vagy ha felhasználókezelés van az alkalmazásba építve, akár perzisztensen eltárolhatjuk egy adott felhasználóról, hogy milyen műveleteket hajtott/hajthat végre.

[7] http://struts.apache.org/2.x/struts2-core/apidocs/org/apache/struts2/components/Token.html

6. Hosszú folyamatok

A többszöri kattintás problémája erőteljesebben ott jelentkezik, ahol a felhasználó egy hosszú és erőforrás igényes műveletet hív meg. Persze alapvetően az interaktív webes alkalmazásokat úgy kell megtervezni, hogy lehetőleg azonnal válaszolni tudjanak minden kérésre, de ez bizonyos esetekben nem biztosítható. Legyen például ez egy lekérdezés, ahol a felhasználó megadja a keresési feltételeket, és válaszra vár, ahol a válasz kiszámítása hosszabb idő. Még rosszabb, ha nem tudjuk előre megjósolni a folyamat időtartamát, sőt az nagy mértékben változhat, azonnal is adhat vissza eredményt, de akár több perc múlva is. A feladat nem az, hogy a felhasználó kétszer ne tudja elindítani a lekérdezést, hanem az, hogyha már elindított egy lekérdezést, akkor ne tudjon egy újat elindítani, vagy ha igen, akkor lehetősége legyen az előző hosszú folyamatának megszakítására.

Abban az esetben, ha nem korlátozzuk a felhasználó lehetőségeit, megteheti azt, hogy egy hosszú folyamat futása közben bezárja a böngészőjét, esetleg újraindítja, majd újra elindít egy hosszú folyamatot. Az első folyamat ilyenkor még nem áll le, hiszen a http protokoll kérés-válasz működésének megfelelően a szerver nem kap értesítést arról, hogy a felhasználó megszakította az oldal letöltését. Ilyenkor a régi szálak még mindig futnak a web-konténeren belül, de kimenetük elveszik, választ csakis az utolsó szál képes a felhasználó böngészőjének visszaadni. Ha a felhasználók nem kapnak azonnal választ, többször is kattinthatnak, és a legrosszabb esetben a szerver össze is omolhat a sok párhuzamos kérés miatt.

A Synchronizer Token technika segíti azt, hogy a felhasználó ne tudjon újra küldeni egy űrlapot, de nem definiálja, hogy hogyan kezeljük a hibát, ha mégis megpróbálja azt. Ráadásul ebben az esetben az előzőekben említettnek megfelelően az első szál végzi a munkáját, egy idő után akár sikeresen be is fejeződhet, csak a felhasználó nem kap erről értesítést, és jogosan gondolhatja, hogy a művelete sikertelen volt.

Erre a problémának a megoldása nagyon sokféle lehet, de mindegyik a párhuzamos programozásra vezethető vissza. A hosszú folyamatot javasolt egy külön szálon végzi, és a beérkezett kéréseket is külön szálak kezelik. Ez utóbbiak között is vannak közbülső szálak, melyeket a felhasználó többszöri kattintásának kiszolgálására indított a web-konténer, illetve van az utolsó szál, mely kapcsolatot tartja a felhasználó böngészőjével. Fő cél, hogy a közbülső szálakat mindig szüntessük meg, mert érdemi munkát nem végeznek, lehetőleg azonnal térjenek vissza.

Összefoglalva a szálak:

- Hosszú folyamatot végző szál
- Közbülső szál, mely érdemi munkát nem végez
- Aktuális szál, mely kiszolgálja a böngészőt

Általánosan az a tévhit terjedt el, hogy J2EE alkalmazásban nem szabad szálakat alkalmazni. Ezt a szabvány nem írja elő, sőt nem is ajánlja a szálak használatának mellőzését. Csupán az EJB rétegben nem szabad szálakat alkalmazni, web rétegben a megfelelő odafigyeléssel lehetséges. A régebbi EJB

2.0-ás szabványban is, ahol még nem szerepelt az EJB timer service (mely időzített funkciók aszinkron végrehajtását teszi lehetővé), időzített funkciók végrehajtására ajánlás volt, hogy a web rétegben induljon el egy szál (esetleg Timer, vagy egy időzítő keretrendszer, pl. a Quartz), és az hívjon be kívülről az EJB rétegbe. Web réteg hiánya esetén operációs rendszerből ütemezett kliens alkalmazást javasolt.

A felhasználó a hosszú folyamatot többféleképpen érzékelheti. Főleg viszonylag rövidebb, de az interaktív felhasználásnál egy kicsit hosszabb folyamatoknál várakoztathatjuk, azaz addig nem kap vissza választ, míg a folyamat le nem fut.

Tényleg hosszabb folyamatok esetén az alkalmazás visszaadhat neki valamilyen választ, hogy a folyamat elindult, és várakozzon. Ilyenkor ajánlott az oldalt automatikusan újratöltődővé tenni, sőt lehetőséget adni arra is, hogy a felhasználó manuálisan is újratölthesse az oldalt. Még szebb megoldás, ha a felhasználó visszajelzést kaphat arról, hogy hol áll a folyamat (százalékban, esetleg előreláthatólag mennyi idő van még hátra). Optimális esetben a felhasználó arra is lehetőséget kap, hogy a folyamatot megszakíthassa.

Vannak olyan esetek, hogy nem lehet előre megmondani, hogy a folyamat hosszú-e vagy rövid, pl. bizonyos adathalmaz esetén egy tárolt eljárás végrehajtása lehet a másodperc töredék része, de más adathalmaz esetén akár több perc. Ilyenkor, ha a folyamat rövid, felesleges a felhasználónak kiírni egy tájékoztató üzenetet, hogy várjon, majd azonnal lefut a folyamat, és visszaadni neki az eredmény oldalt. Ekkor az előző két eset kombinációját kell választani, azaz egy külön szálban elindítjuk a folyamatot, ami lehet rövidebb és hosszabb is, majd a felhasználót kiszolgáló szálat várakoztatjuk 1-2 másodpercig (ennek kiválasztása függhet az alkalmazás jellegétől, a felhasználók hozzáállásától, stb.). Ha ezalatt lefutott a folyamat, azonnal a választ adjuk vissza, ha nem futott le, akkor a várakoztató oldalt.

Annak tárolására, hogy a felhasználó már elindított egy folyamatot, és a folyamat állapotát jelző objektumot (egyszerűbb esetben azt, hogy befejezte-e már, bonyolultabb esetben a százalékos készültséget, vagy bármilyen tájékoztató információt) legegyszerűbb a felhasználó munkamenetében tárolni, akár az egész szál objektumot, ezen értékeket pedig a szál adattagjaiként definiálva. Ha a felhasználó meg is szakíthatja a műveletet, akkor ide érdemes elhelyezni egy flag-et, ami jelzi, hogy felhasználó megszakította-e a műveletet.

Ennek a technikának egy változata, hogy az első kéréshez tartozó szálat engedjük, hogy végrehajtsa a hosszú folyamatot, és a kérés paramétereit, valamint a választ eltároljuk a felhasználó munkamenetében, amint az előállt. Így a további kérések esetén, ha ez első szál még nem végzett, akkor egy tájékoztató oldalt adunk vissza, ha végzett, akkor a munkamenetből elővesszük az ott előállt eredményt.

A hosszú folyamat kezelésére nem egyszerű példát hozni, ugyanis rengeteg kérdés és merül fel ezzel kapcsolatban, melyeket alkalmazásonként másképp kell kezelni. Ezen problémák megoldása nélkül azonban a példa használhatatlan, megoldásával meg kezelhetetlenül bonyolultá válik. Ezen problémák a következők.

- Amennyiben egyszerű szálakat indítunk minden esetben, nagy terhelés esetén túl sok szál keletkezhet, melyek az alkalmazásszerver összeomlását idézhetik elő.

- Amennyiben a szálak számát kézben akarjuk tartani, pool-t kell alkalmazni. Azonban ilyenkor valamilyen komponensnek el kell indítania a pool-t, és megfelelően le is kell azt állítania. A szál indításakor kezelni kell, ha a pool megtelt, vagy éppen leállás alatt van.

- Nem lehet egyszerűen a hosszú műveleteket sessionben tárolni. Egyrészt a session-ben csak szerializálható objektumokat lehet tartani, ami ebben az esetben nem értelmezett. Így valamilyen külön tárolót érdemes kialakítani.

- A hosszú folyamatoknak érdemes túlélniük egy szerver újraindítást is. Itt nem csak arra kell figyelni, hogy a folyamatot lementsük, hanem annak jelenlegi állapotát, valamint visszatöltéskor újra is legyen az indítva. Ugyanígy élje túl a véletlen, nem tervezett leállást is.

- Cluster-es működés esetén figyelni kell a lábak közötti szinkronizációra.

- Ahogy fentebb említettem, egy hosszú folyamatról jó, ha a felhasználó információt kap, hogy hogy áll, és akár meg is szakíthatja azt. Esetleg még szüneteltetheti és újraindíthatja azt.

Az aszinkronitást ezen kívül nem csak szálakkal valósíthatjuk meg, hanem JMS technológiával (,végső soron ezek is szálakra vezethetők vissza, csak az alkalmazásszerver maga kezeli azokat). Ez sokkal robosztusabb megoldás, azonban nagyobb infrastruktúrát is igényel.

Természetesen erre is van egy J2EE tervezési minta, amely a hosszú folyamatok elfedésére szolgál, neve Service Activator. Feladata, hogy egy aszinkron réteget épít be a kliens - a szolgáltatást igénybe vevő -, valamint a szolgáltatás közé. A felhasználót kiszolgáló komponens akár több rétegen keresztül egy üzenetet dob egy sorba, majd azonnal visszatér hozzá a vezérlés, és kiszolgálhatja a felhasználót (általában ez egy üzenet, hogy a kérését beütemeztük, amikor lehetőség adódik rá, el lesz végezve). A szolgáltatás oldalon a sorra regisztrált figyelőt (egy Message Driven Bean) értesíti az alkalmazásszerver (nem definiált, hogy mikor, amikor erőforrás van rá), hogy a sorba új üzenet érkezett, és ez már szinkron hívja a hosszú folyamatot. Látható, hogy ez a megoldás kapcsolatban van a Half-Sync/Half-Async tervezési mintával (([HalfSyncAsync])), ahol az egyik oldalon a felhasználók kéréseket intéznek a rendszer felé, és azonnali választ várnak (szinkron), a másik oldalon bizonyos szolgáltatások aszinkron módon működnek. Egyszerűsíteni ennek kezelését sorok bevezetésével lehet. Így megmarad a szinkronitás, az aszinkronitás is, és a kettő közötti átjárhatóságot a sorokon alapuló réteg biztosítja.

7. AJAX

Amint látható, a legtöbb webes alkalmazásokat fejlesztő, valamint webes keretrendszereket fejlesztő és használó szakember szembesült a problémával, hogy mi történik akkor, ha egy felhasználó nem az általunk megálmodott módon használja az interaktív webes alkalmazást, hanem többször klikkel, valamint szabadon használja a böngésző navigációs adottságait. Egyre több technika van ezen problémák megoldására, és napjainkban szerencsére ezek egységes elnevezéseket kapnak, és egyre több platformon, programozási nyelvben elérhetővé válnak, valamint a webes keretrendszerekbe is integrálják őket.

Napjaink bűvszava az AJAX sem mentes ezektől a problémáktól, de azok itt pont fordítva jelentkeznek. Az AJAX Jesse James Garrett szerint ([JamesGarett2005]) szerint nem egy technológia, hanem már

meglévő technológiák felhasználása egy új, meglehetősen hatékony módon. Az AJAX (Asynchronous JavaScript + XML rövidítése) a következő technológiákat foglalja magában:

- Standard adathordozó és megjelenítésre használatos szabványok XML, XHTML, CSS
- Document Object Model
- XMLHttpRequest objektum a használata szerverrel történő aszinkron kommunikációra
- JavaScript az egész összefogására

Az AJAX alapvetően úgy működik, hogy amennyiben a felhasználó valamilyen műveletet végez az oldalon, az nem generál kötelezően egy új oldalletöltést, hanem a háttérben, a XMLHttpRequest objektum használatával hívódik meg a szerver oldal, és amit az visszaad, az jelenik meg az oldalon, az oldal frissítése nélkül. Ez a mi szemszögünkből azt eredményezi, hogy mivel nincs új oldalletöltés, nem jelenik meg újabb oldal, és nem kerül be a böngésző előzményei közé, és nem lehet az oldalak között navigálni. Ez az egyik szempontból jó, mivel nem kell felkészülnünk a szabad navigáció okozta problémákra, másrészről elég kényelmetlen, hiszen a felhasználónak nem adunk kényelmi lehetőséget a saját szájíze szerinti böngészésre, esetleg egy bizonyos közbülső oldal Kedvencek közé történő tárolására. Természetesen AJAX használata esetén is lehetőség van a szabad navigáció biztosítására. Ehhez egyrészt a felhasználói aktivitásoknak megfelelően különböző állapotokat kell definiálni, URL-ekhez kötni, majd elhelyezni a böngésző előzményei között (ez JavaScript-tel egyszerűen megoldható). Amikor a felhasználó szabad navigációt használ, az URL-ből meg kell állapítani az állapotot, és azt kell a felhasználónak előállítani. Ebben az esetben viszont ugyanúgy figyelni kell a felhasználó szabad navigációja okozta problémákra, mint hagyományos webes alkalmazás esetében.

A hosszú folyamatok kezelése AJAX esetén szintén fokozottan előjön. Egyrészt egy hosszú folyamat állapotát érdemes mindenképpen AJAX-szal jelezni, hiszen így nincs szükség oldal újratöltésre. Másrészt a http ugyan kérés-válasz alapú, de tipikusan ilyen esetekben van szükség arra, hogy a szerver oldal szóljon a kliens oldalnak. Ezt un. Comet technológiával[8] szokták megvalósítani. Ez azt jelenti, hogy a böngésző nyitva tart egy http kapcsolatot a szerver oldal felé, és az nem válaszol, hanem addig tartja a kapcsolatot, amíg nem akar valamilyen információt a kliens felé átvinni. Ez egy remek trükk, azonban rengeteg problémával jár. Egyrészt nyitva kell tartani egy http kapcsolatot. Ez egyrészt elvesz a böngészőn belül egy kapcsolódási lehetőséget, így ezen a kapcsolaton statikus tartalom, mint css, kép, JavaScript fájl nem közlekedhet (ennek a böngésző szabta korlátok miatt van jelentősége). Másrészt a tűzfalak sem szeretik a tétlen szálakat. Harmadrészt az alkalmazásszervert is terhelhetik, hiszen általában minden kéréshez egy külön szál tartozik. Az tűzfalakat finomhangolással ki lehet játszani, polling-olni kell, azaz nem egy hosszú kapcsolatot, hanem több, rövidebb, de normál http kéréshez képest hosszabb (timeout alatti) kapcsolatot kell használni. Másrészt Java oldalon pont a Servlet 3.0 szabvány vezette be az aszinkron feldolgozás fogalmát. Sajnos ez sem oldja meg teljeskörűen a problémát, hiszen csak annyit old meg, hogy egy ilyen kérés ne használjon fel feleslegesen egy alkalmazásszerver szálat, hanem csak a TCP/IP kapcsolat maradjon nyitva. Elméletileg a HTML 5 erre is megoldást fog nyújtani a WebSocket API-val.

[8] http://en.wikipedia.org/wiki/Comet_%28programming%29

Irodalomjegyzék

[Thomason2002] Thomason, Larisa. Avoid Duplicate Form Submissions, NetMechanic Webmaster Tips[9].

[bib.Jouravlev2004] Jouravlev, Michael. Jouravlev2004Redirect After Post[10].

[CoreJ2eePatterns] Alur, Deepak. Crupi, John. Malks, Dan. Core J2EE Patterns[11]. Best Practices and Design Strategies. 2003. 2. ISBN: 0131422464.

[DevelopersAlmanac] Chan, Patrick. The Java™ Developers Almanac 1.4, Volume 1. Examples and Quick Reference. 2002. 4. ISBN: 0201752808.

[Senior2009] Senior, Richard. Simple Synchronizer Token with Spring MVC[12].

[Reumann] Reumann, Rick. Using Token in Struts[13].

[Deabill2008] Deabill, Jason. Firefox 3 and the "cache-control" header[14].

[HalfSyncAsync] C. Schmidt, Douglas. D. Cranor, Charles. Half-Sync/Half-Async -- An Architectural Pattern for Efficient and Well-structured Concurrent I/O[15].

[JamesGarett2005] James Garett, Jesse. Ajax: A New Approach to Web Applications[16].

[9] http://www.netmechanic.com/news/vol5/html_no16.htm
[10] http://www.theserverside.com/news/1365146/Redirect-After-Post
[11] http://www.corej2eepatterns.com/index.htm
[12] http://explodingjava.blogspot.com/2009/03/spring-mvc-synchronizer-token.html
[13] http://www.learntechnology.net/content/struts/struts_token.jsp
[14] http://blogs.imeta.co.uk/JDeabill/archive/2008/07/14/303.aspx
[15] http://www.cs.wustl.edu/~schmidt/PDF/PLoP-95.pdf
[16] http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications