A. függelék
Webalkalmazások

Az alábbiakban röviden bemutatjuk a Java *szervleteket* és a hozzájuk kapcsolódó *JSP (Java Server Pages)* oldalakat. Ezen kiszolgáló- és platformfüggetlen technológia a CGI esetében felmerülő összes lényeges problémára (lásd az 1.1.4.1. szakaszt) megoldást nyújt.

Az elmondott technológia Java-alapú, de megjegyzendő, hogy a Microsoft cég szerveroldali megoldása, az *ASP (Active Server Pages)* nagyon hasonló elgondolásokat és ötleteket használ a webalkalmazások fejlesztése kapcsán. Az ASP-ről bővebben a [82] könyvben olvashatunk.

A.1. Szervletek

A szervletekre tekinthetünk úgy, mint a CGI-k Java-alapú alternatívájára. Valójában egy szervlet egy Java-osztály, melyet a kiszolgáló példányosít és futtat. A szervletek közönséges Java-programok, ezért tetszőleges Java-könyvtárat, Java nyelvi konstrukciót használhatnak.

A Java platformfüggetlensége, hordozhatósága és biztonsága hatalmas előny. Ugyanúgy, ahogy egy Java-programot valóban változtatás nélkül futtathatunk például Windows, Linux vagy éppen MaxOS X alatt, a Java-szervletek is gond nélkül futnak a fejlesztőitől különböző platform esetén. A JVM (*Java Virtual Machine* – Java virtuális gép) használata miatt a Java-programok kontrollált környezetben futnak, szó sem lehet puffertúlcsordulásból adódó biztonsági problémákról.

A szervlet API (*Application Programming Interface* – alkalmazói programinterfész), ahogyan azt hamarosan látni fogjuk, számos kényelmi szolgáltatást nyújt. Például a menetek (*session*) kezelése, a kérési paraméterek feldolgozása automatikus történik.

Érdemes eloszlatni azt a tévhitet, hogy a szervleteket használó dinamikus oldalak, mivel Java-alapúak és minden Java alkalmazás örökletesen lassú, lassan töltődnek be és lassan reagálnak a felhasználói kérdésekre. Számos mérés és gyakorlati tapasztalat bizonyítja, hogy igazán nagy terhelés esetén ennek éppenhogy az ellenkezője igaz. Mindemellett, nem tagadjuk annak az állításnak az igazságát, hogy kis feladatok elvégzéséhez egy Java-szervlet túlságosan nagyágyú és sokszor körülményes. Nagy alkalmazások esetében azonban a befektetett munka bőségesen megtérülhet.

A.1.1. Bevezetés

Az alábbiakban megírjuk és futtatjuk első, működőképes szervletünket. A szervletek futtatásához szükségünk van egy megfelelő webkiszolgálóra (szervletkonténerre). Ilyen lehet az Apache Tomcat, amely a szervlettechnológia referenciaimplementációja, az iPlanet Web Server, amelyet a Sun fejleszt, a Jigsaw, amely a W3C saját webkiszolgálója vagy éppen az IBM WebSphere, illetve a BEA WebLogic, csak hogy a legfontosabbakat említsük.

Első szervletünk feladata, hogy egy olyan szolgáltatást nyújtson, amely segítségével weben keresztül összeadhatunk két számot. Ehhez első lépésként szükségünk van egy űrlapra (az A.1. ábra), amelyben bekérünk két számot. A vezérlést ezután egy szervlethez (az A.2. ábra) továbbítjuk (pontosan úgy, mint egy közönséges CGI-alkalmazás esetén, lásd az 1.1.2. szakaszt), amely által visszaadott válaszoldal a két szám összegét tartalmazza.

A megadott űrlap esetén fontos észrevenni, hogy ez egy közönséges HTML-űrlap, pontosan ugyanolyan felépítésű, mint amilyet például az 1.1.3. szakaszban láthattunk. A két beviteli mező neve rendre `szam1` és `szam2`, ezekben adhatjuk meg a legfeljebb 3 számjegyből álló számokat. A `submit` típusú `Elküld` gomb megnyomása után a kérést GET metóduson keresztül továbbítjuk a megnevezett szervletnek.

```
<html>
  <head>
   <title>Összeadó példa</title>
   <meta http-equiv="content-type"
         content="text/html; charset=utf-8"/>
  </head>
  <body>
   <form action="http://wald.hu/addossze.add" method="get">
    1. szám: <input type="text" size="3" name="szam1"></br>
    2. szám: <input type="text" size="3" name="szam2"></br>
    <input type="submit" value="Elküld">
   </form>
  </body>
</html>
```

A.1. ábra. Első szervletünkhöz tartozó űrlap

Az A.2. ábrán látható szervlet egy közönséges Java-program. Az általunk definiált `Osszeado` osztályt a `HttpServlet` osztályból származtatjuk és felüldefiniáljuk annak `doGet` metódusát. Ez a metódus hivatott a GET formájú kérések kezelésére. A `doGet` metódusnak két paramétere van, a `request` objektum minden fontos információt tartalmaz a kérésről, a `response` objektum pedig a válaszunk megfelelő megadására szolgál. Fontos, hogy a szervlet kódjában tetszőleges Java-kódot felhasználhatunk, amire persze szükségünk is van, ha az elvégzendő feladat bonyolultabb, mint egy összeadás.

Szervletünk most azonban ezt a végtelenül egyszerű feladatot oldja meg. Elkéri a `szam1` és a `szam2` paraméterek értékét (ezeket nem számként kapjuk meg, ezért először át kell azo-

kat konvertálni), majd elkészít egy válaszoldalt, amelyben a `szam1+szam2` kifejezés értékét helyezi el. A választ tartalmazó HTML-oldalt teljes egészében mi állítjuk elő, megfelelő fejjel, karakterkódolással, törzzsel stb. Amennyiben az lett volna a feladat, hogy egy számról döntsük el, hogy prímszám-e, megtehettük volna azt is, hogy két, előre elkészített statikus weblap (igen/nem) közül adunk vissza egyet. Ebben az esetben a szervletünk maga nem tartalmazott volna HTML-kódot.

Kész szervletünket a szokásos módon fordítjuk le a Java-fordító (`javac`) segítségével, a kapott `.class` állomány már maga a futtatható szervlet. Ezt a webkiszolgáló megfelelő könyvtárába kell elhelyezni a webalkalmazásunk többi állományával együtt.

```
package hu.cs.bme.szemweb;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Osszeado extends HttpServlet {
 public void doGet(HttpServletRequest request,
                   HttpServletResponse response)
                   throws ServletException, IOException {

  int sz1=Integer.parseInt(request.getParameter("szam1"));
  int sz2=Integer.parseInt(request.getParameter("szam2"));

  PrintWriter out = response.getWriter();
  response.setContentType("text/html");

  out.println("<html>");
  out.println("<head><title>Eredmény</title>
   <meta http-equiv=\"Content-Type\" content=\"text/html;
   charset=ISO-8859-2\"/></head>");
  out.println("<body>");
  out.println("Az összeadás eredménye: "+(sz1+sz2));
  out.println("</body>");
  out.println("</html>");
 }
}
```

A.2. ábra. Első szervletünk – adjunk össze két számot!

A szervlet működéséhez szükségünk van azonban még arra, hogy összekössük az űrlapot és a szervletet, azaz hogy megmondjuk a webkiszolgálónak, hogy bizonyos kérések esetén mely szervletet kell használni. Erre szolgál a webalkalmazásunk *telepítésleírója*, amely fontos információkat ad a webalkalmazásunkról a szervletkonténernek.

A fentieknek megfelelően a telepítésleíró legfontosabb feladata, hogy megadja, mely hívási minta esetén mely szervlet kerüljön meghívásra. Láthatjuk az A.1. ábrán, hogy szervletünk megnevezésére a `http://wald.hu/addossze.add` címet használtuk. Valójában a `wald.hu` kiszolgálóhoz intéztünk egy HTTP-kérést annak érdekében, hogy megkapjuk az `addossze.add` állományt.

Telepítésleírónk valójában azt specifikálja, hogy minden `.add` kiterjesztésű állományra vonatkozó kérés esetén az összeadást elvégző szervletet kell meghívni (az A.3. ábra). Megadhattunk volna más mintát is, például azt, hogy csak és kizárólag a `addossze.add` kéréseket kezelje így. A telepítésleíró meghatározza továbbá, hogy pontosan melyik Java-osztály is a kérdéses szervlet, valamint megadja azt is, hogy mely oldal legyen webalkalmazásunk nyitó oldala.

```
<?xml version="1.0" encoding="ISO-8859-2"?>
<web-app>
 <display-name>Összeadó</display-name>
 <description>Példa szervlet</description>

 <servlet>
  <servlet-name>osszeado</servlet-name>
  <servlet-class>hu.cs.bme.szemweb.Osszeado</servlet-class>
 </servlet>

 <servlet-mapping>
  <servlet-name>osszeado</servlet-name>
  <url-pattern>*.add</url-pattern>
 </servlet-mapping>

 <welcome-file-list>
  <welcome-file>osszeado.html</welcome-file>
 </welcome-file-list>
</web-app>
```

A.3. ábra. Első szervletünk telepítésleírója

A.1.2. Szervletek életciklusa

Egy konkrét szervlet valójában egy közönséges Java-objektum. Egy kérés teljesítése a szokásos metódushívásnak felel meg.

Előző példánkban, amennyiben két szám összegére vagyunk kíváncsiak, végeredményben szervletünk `doGet` metódusát hívjuk meg. Ennek megfelelően, egy szervlet által megvalósított funkció hatékonyan érhető el, de ami a legfontosabb, hogy a szervletpéldány az egyes hívások között megmaradhat. Egy szervlet így megnyitva tarthat különböző állományokat, adatbázis-kapcsolatokat, megjegyezheti bizonyos bonyolult számítások eredményét

abban a reményben, hogy a közeljövőben érkező hasonló kérdésekre így majd gyorsabban tud válaszolni.

Valójában a szervletek életciklusa csak nagyon lazán meghatározott. Egy webkiszolgálónak mindössze a következő lépéseket kell támogatni:

1. a szervlet inicializálása és példányosítása
2. az esetleges klienskérések esetén a szervlet megfelelő metódusának meghívása
3. a szervlet leállítása és szemétgyűjtés

Ad absurdum tehát az is elképzelhető, hogy a szervletkonténer egy új kérés esetén létrehoz egy új szervletpéldányt, kiszolgálja segítségével a kérést, majd közvetlenül ezután le is állítja azt, hasonlóan az alap CGI megoldáshoz. A valóságban azért nem ez történik, mert ilyen hozzáállással a CGI-hez hasonló hatékonysági problémák jelentkeznének, és a piaci verseny egyszerűen kiszorítaná az ilyen webkiszolgálókat. Az alábbiakban feltételezzük, hogy a kérések között a szervletpéldány életben marad.

A leggyakoribb esetben egy szervletpéldány létezik, amely szempontjából az egyes kliensektől érkező kérések egy-egy szálnak felelnek meg. Minden olyan esetben, amikor szálakkal van dolgunk, nagyon körültekintően kell eljárnunk. Amennyiben a `doGet` metóduson belül csak lokális változókat használunk, nincsen mitől tartanunk. Akkor azonban, ha *példány-* vagy *osztályváltozókra* is szükségünk van (azért, mert szervletünk működéséhez szükséges, hogy az egyes hívások között bizonyos állapotokat megőrizzünk), gondjaink lehetnek.

Az elmondottakra példa egy olyan szervlet, amelyik egyrészt számolja, hogy hányszor hívták meg, másrészt vissza is adja ezt az értéket. Egy ilyen szervlet belsejéből származhat az alábbi kódrészlet, ahol `szamlalo` egy példányváltozó és mivel az egyes kérések között a szervletpéldány életben marad, valóban a helyes értéket fogja tartalmazni:

```
...
szamlalo++;
out.println("Teljesített kérések száma: "+szamlalo);
...
```

Sajnos azonban előfordulhat az az eset, hogy két szál egyszerre tartózkodik a `doGet` metódus belsejében, és az élet úgy hozza, hogy az operációs rendszer az alábbi sorrendben hajtja végre az utasításokat:

```
szamlalo++;          // első szál
szamlalo++;          // második szál
out.println("...");  // első v. második szál
out.println("...");  // második v. első szál
```

Szerencsétlen módon tehát kétszer növeltük meg *ugyanazt* a példányváltozót és ezt a kettővel megnövelt értéket adjuk vissza a klienseknek. Persze jelen esetben ez nem okoz valódi problémát, legfeljebb két ember a világ két pontján egyszerre látja ugyanazt a számot. Sokszor azonban az ilyen jellegű *szinkronizációs* probléma „életveszélyes”, amennyiben az adott példányváltozó fontos információt tárol.

A szálak kapcsán látott szinkronizációs problémára megoldásként elsőként felmerülhet, hogy használjunk lokális változókat példányváltozók helyett. Ezzel azonban éppen azt a lehetőséget veszítjük el, hogy szervletünk megtartsa állapotát a kérések között. Az igazi megoldás a Java nyelv által támogatott szinkronizációs lehetőségek használata, amelyekkel

bizonyos kóddarabokról megkötjük, hogy azok csak atomi, azaz szétbonthatatlan módon kerülhetnek végrehajtásra.

Itt is óvatosnak kell lennünk, mert minél nagyobb kóddarabot helyezünk védelem alá, annál többet veszíthetünk a hatékonyságból. Szélsőséges esetben, amennyiben a teljes `doGet` metódust védetté nyilvánítjuk, szervletünk egyidőben egyetlen kérés kiszolgálását végezheti csak el (képzeljük el, hogy mennyi hasznos CPU-időt fecsérlünk el, ha ilyenkor a metódus belsejében például egy adatbázis-lekérdezés eredményére várunk, és nem is használjuk a processzort – ezalatt más szervletek kiszolgálását végezhetnénk el).

Azt is tudnunk kell, hogy hasonló esetekben a szervletkonténer hatékonysági okokból dönthet úgy, hogy új szervletet példányosít. Ez annyival bonyolítja tovább a dolgokat, hogy a példányváltozók elkülönülnek az osztályváltozóktól. Ennek megfelelően az A.4. ábrán látható szervlet, akkor is, ha szinkronizálva fut, különböző értékeket adhat eredményül a „Kérések száma” és az „Összes kérések száma” vonatkozásában.

Összességében azt kell mondanunk, hogyha a feladatunk megköveteli, hogy az egyes kérések között állapotot tároljunk, akkor érdemesebb ezt osztályváltozókban, mintsem példányváltozókban megtenni. Mindkét esetben lesznek ugyan szinkronizációs problémáink, de osztályváltozók esetén legalább képesek vagyunk fenntartani egy globális állapotot. Példányváltozók használatával ezt nagyon nehéz megoldani, hiszen ehhez az kellene, hogy az egyes szervletpéldányok folyamatosan tájékoztassák egymást a példányváltozóikban bekövetkezett változásokról.

További érdekes dolog az, hogy nemcsak a szervlet maradhat/marad meg az egyes hívások között, hanem a szervlet által elindított szálak is (például azok, amelyeket az inicializációs fázisban indított el). Ennek megfelelően egy szervlet a hívásai *között* is végezhet hasznos munkát. Elsőre ez talán értelmetlennek tűnhet, amennyiben a szervletet pusztán egy bizonyos kérdésekre jól meghatározott válaszokat szolgáltató CGI-alkalmazásnak fogjuk fel. Sokszor azonban ugyanarra a kérdésre nem ugyanazt a választ adjuk, például azért, mert bizonyos háttérben futó számítások eredményei már ismertek, és ezeket felhasználva pontosabban válaszolhatunk. Példa erre egy olyan szervlet, amely egy háttérben lévő adatbázist elemez és folyamatosan frissíti statisztikáit, amelyet a kliensek lekérdezhetnek. Teljesen nyilvánvaló, hogy a feladat kivitelezhetetlen lenne, amennyiben a szervlet a kliens kérésének pillanatában kezdené el az adatbázis vizsgálatát.

Végezetül érdemes megjegyezni, hogy a szervletet sokszor előre létrehozzuk, azaz elvégezzük a példányosítást. Ilyenkor az első kérés kiszolgálásakor sem kell számolni a példányosítás okozta többletköltséggel (egy példány inicializálásakor megnyithat távoli kapcsolatokat, amelyek felépítése meglehetősen időigényes is lehet). A gyakorlatban ez úgy zajlik, hogy a webkonténer indulásakor példányosítja az általa kezelt szervleteket.

A.1.3. Szervletkonténerek fajtái

A szervletkonténerek túlnyomó többsége egyetlen Java virtuális gépet használ a szervletek futtatásához. Ennek hatalmas előnye, hogy így lehetővé válik a hatékony és biztonságos adatmegosztás a szervletek között. Amennyiben maga a webkonténer is Java nyelven íródott (például Apache Tomcat), akkor a szervletek a konténerrel egy virtuális gépen belül futnak. Ilyenkor a szervletek metódusainak meghívása valóban nagyon hatékony.

Amennyiben a konténer nem Java nyelven íródott, és nem multiprocesszes (azaz mindössze egy folyamat van, az egyidejű működést szálak segítségével oldja meg), a Java virtuális

```
public class Szamlalo extends HttpServlet {
  static int oszamlalo = 0;
  int szamlalo = 0;

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
                    throws ServletException, IOException {
   ...
   szamlalo++;
   oszamlalo++;
   out.println("Kérések száma: "+szamlalo);
   out.println("Összes kérések száma: "+oszamlalo);
  }
}
```

A.4. ábra. Osztályváltozók szervletekben

gépet be lehet ágyazni a konténer folyamatába, és taszkváltás nélkül el lehet érni. Ugyanezt nem lehet megtenni egy olyan konténer esetében, amelyik folyamatokat (is) indít a kérések kezelésekor. Ilyenkor a Java virtuális gép egy külön folyamatban fut, amelyre a váltás már meglehetősen költséges.

A.1.4. Menetkezelés

A szervletkonténer automatikusan végzi a menetek kezelését. Amennyiben a kliensoldalon engedélyezve vannak a *piték* vagy *sütik* (angolul: *cookies*), akkor azt használja. Piték hiányában a kérésbe kódolja bele a menet információkat. Ezzel a szervletprogram írójának nem kell törődnie.

A szervlet belsejében a menetet a kérési objektum `getSession()` metódusával kérhetjük el.

```
HttpSession session = request.getSession();
```

A kapott menetobjektumban a `setAttribute()` metódussal tetszőleges Java-objektumot tárolhatunk; ennek egy kulcsnevet és magát a tárolandó objektumot kell megadnunk. Az adott kulcs alatt tárolt objektumot a `getAttribute()` metódussal kaphatjuk vissza.

```
session.setAttribute("user",
                     new Silk_user(name,passwd,profile));
...
Silk_user user = (Silk_user)session.getAttribute("user");
```

A menetekhez lejárati idő is tartozik (ezt láthatjuk akkor, amikor bankunk webes ügyfélszolgálata bizonyos inaktívitási idő után újra megkér minket, hogy azonosítsuk magunkat), amelyet több helyen is állíthatunk, többek között az alkalmazásunk telepítésleírójában is.

A.2. Java Server Pages

A szervletek esetében a biztonság, hatékonyság és kényelem magas fokon valósul meg. Sajnos azonban az üzleti logika és a megjelenítés nem válik szét kellő mértékben. Új felület létrehozása, a felület változása esetén módosítani kell a szervlet forráskódját, sőt, a változások átvezetéséhez újra is kell fordítani azt. Ez a megoldás megnehezíti az üzleti logika felhasználását más környezetben. Elképzelhető, hogy egy nem webes grafikus felületen is el szeretnénk érni rendszerünket, sőt, az is, hogy be szeretnénk ágyazni más rendszerekbe mindenféle megjelenítés nélkül. Amennyiben az üzleti logika élesen különválik a megjelenítéstől, ez sokkal könnyebben kivitelezhető, mint amikor össze vannak nőve.

További hátrány, hogy a rétegek (üzleti logika, megjelenítési) összemosása megnehezíti a csapatmunkát is. Gondoljunk arra, hogy a szervletek írásához alapvetően más emberek értenek, mint akik honlapot készítenek (akik amúgy is sokszor nem kézzel teszik ezt, hanem különféle vizuális szerkesztőeszközökkel). Egy bank esetében a mögöttes logika, a tranzakciók biztonságos végrehajtása, az ügyfélazonosítás, a banki adatbázis lekérdezései mindmind olyan dolgok, amelyek nem kapcsolódnak a megjelenítés módjához, ezeket mindig ugyanúgy kell végezni. Rémálom lenne, ha egy stilisztikai változtatás miatt hozzá kellene nyúlni az egész rendszer forráskódjához azért, mert ott generáljuk a kimenő HTML-oldalt.

Meg kell jegyezni, hogy ugyanakkor egy szervlet alkalmas arra, hogy a munka lényegi részének elvégzése után előre elkészített oldalak, mint lehetséges válaszok közül válassza ki a megfelelőt. Ilyenkor a rétegek elválasztása kellően megoldott. Sajnos azonban ez sokszor nem járható út, gondoljunk arra, hogy a számösszeadós esetben nem lehet előre elkészíteni minden lehetséges válaszoldalt. Az elgondolás mindenesetre nagyon jó, a szervlet foglalkozzon a lényegi résszel, majd adja tovább a vezérlést egy oldalnak, amely képes megjeleníteni az eredményt.

Kicsit továbbgondolva látható, hogy az is igaz, hogy a legtöbb esetben az oldalnak csak egy bizonyos része tartalmaz dinamikusan generált információt (például egy adott helyen a számítás/adatbázis-lekérdezés eredményét), a maradék változatlan statikus kód. Ezt úgy is felfoghatjuk, hogy valójában egy *sablonra* van szükségünk, amelyet előre elkészíthetünk úgy, hogy bizonyos részeit kitöltetlenül hagyjuk. Ide kerül majd a dinamikus tartalom.

Pontosan ezt az elvet követi a *Java Server Pages (JSP)* technológia. A JSP-oldalak első megközelítésben közönséges HTML-oldalak, amelyek néhány helyen speciális elemeket tartalmaznak. Ezen elemek feladata a dinamikus tartalom generálása, azaz a sablonunk hiányos részeinek kitöltése. Megjegyezzük, hogy a szabad szoftver világában népszerű PHP is hasonló ötleten alapul.

Első példánk az A.5. ábrán látható. A példa egy közönséges HTML-oldalnak tűnik, de bizonyos helyeken JSP-elemeket használunk. Például a honlap törzsében a `<%` és `%>` jelek között Java-kód szerepel. Azt is láthatjuk, hogy a kódrészlet „gyanús” abban az értelemben, hogy nagyon hasonlít az A.2. ábrán látott szervlet kódjához. A `request` objektumot ott is arra használtuk, hogy információkat tudjunk meg az aktuális kérdésről, például kinyerjük a kérési paramétereket. Jelen esetben, ha ilyen paraméterünk nincsen, a `Kedves ismeretlen!` szöveggel üdvözöljük a látogatót. Amennyiben van `nev` nevű paraméterünk (mert például honlapunkat `http://...honlap.html?nev=Gergo` alakban értük el), akkor viszont nevén szólítjuk. A dinamikus tartalmat az előbbiekhez hasonlóan az `out.println("...")` metódushívással generáljuk. Végül minden egyes kérés esetén beillesztjük az oldalunkba a `new java.util.Date().toString()` kifejezés értékét, amely az aktuális időt írja ki.

Bevezető példánk jól illusztrálja a sablonelvet. Amennyiben ezt a weboldalt egy kliens elkéri a szervertől (webkonténertől), akkor bizonyos részek dinamikusan generálódnak, maga a keret azonban statikus.

```
<html>
  <head>
    <title>Első JSP példa</title>
    <meta http-equiv="..."
          content="text/html; charset=utf-8"/>
  </head>
  <body>
   <h1>
    <% if (request.getParameter("nev") == null) {
      out.println("Kedves ismeretlen!");
    } else {
      out.println("Szia "+request.getParameter("nev")+"!");
    }
    %>
   </h1>
   Az aktuális idő: <%= new java.util.Date().toString() %>
  </body>
</html>
```

A.5. ábra. JSP-elemeket is tartalmazó weboldal

A.2.1. Célok, felépítés

A JSP-technológia egyik célja a szervletekkel történő minél szorosabb együttműködés. Ez teljes mértékben megvalósul, mert valójában egy JSP-oldal *szervletté fordul*. A JSP-lapon található statikus elemekből `out.println("...")` hívások lesznek, a beillesztett Java-kódok egy az egyben bekerülnek a szervletbe stb. A fordítás során automatikusan készül el az osztálydeklaráció, inicializálódik az `out` objektum. A JSP-lap az első elkéréskor vagy a webkonténer indulásakor fordul le szervletté, beállításoktól függően. A szervletből áll elő a JVM által már értelmezhető bináris formátum. Miután ez megtörtént, a kérésekre való reagálás hatékony.

A JSP fő célja tehát az üzleti logika és a megjelenítés különválasztása, amihez a sablonkitöltés elvét használja fel. A sablonban lévő kitöltésre, beillesztésre váró részeket *JSP-elemeknek* hívjuk. Három alapvetőa a JSP-hez kapcsolódó elemkategória létezik.

- *szkriptelemek*
- *direktívák*
- *akcióelemek*

A szkriptelemek segítségével Java-kódot illeszthetünk be oldalunkba. A Java-kód több, előre definiált objektumot is használhat, mint például a `request`, `response`, `session`, `out` stb.

Többféle szkriptelemet különböztetünk meg. Egy *JSP-szkripletet* a <% és %> jelek határolnak, a közrefogott Java-kód a JSP-szervlet fordítása során egy az egyben kerül be a szervlet kiszolgáló metódusába. JSP-szkripletre láthattunk példát az A.5. ábránkon. A *JSP-deklarációk* szintén változtatás nélkül kerülnek át a szervletbe, csak nem a kiszolgáló metódusba, hanem azon kívülre, a szervlet osztály törzsébe. A deklarációkat a <%! és %> jelek határolják. Az alábbi deklaráció egy példányvátozót definiál és inicializál:

```
<%! private int szamlalo = 0; %>
```

Az utolsó fajta JSP-szkriptelem a *JSP-kifejezés*, amelyet a <%= és %> jelek határolnak. A JSP-kifejezés egy Java-kifejezés, amelynek *értékét* illeszti be a webkiszolgáló a végső weblapba. Pontosabban, a kifejezés értékéül kapott objektumot a webkonténer a Java `toString()` metódusával konvertálja kiírható alakba. Az A.5. JSP-lapon, az aktuális idő kiírásakor, láthatunk példát JSP-kifejezés használatára.

A JSP direktívák midegyikét a <%@ és %> jelek határolják, általános felépítésük a következő.

```
<%@ direktívanév attribútumnév-1=érték-1
                 attribútumnév-2=érték-2
                 ...
                 attribútumnév-k=érték-k %>
```

A direktívák a szervlet egész működésére hatással lévő beállításokat definiálnak. Például a `page` direktívanév segítségével megadhatjuk, hogy a szervlet mely csomagokat importálja (hiszen lehet, hogy például az egyik szkripletben nem hagyományos osztályt használunk), mi legyen az előállítandó lap MIME-típusa, mekkora puffert használjon a `PrintWriter` objektum.

A JSP-akcióelemek ismertetésével a következő rész foglalkozik.

A.2.2. JSP-akcióelemek, JavaBean-komponensek

Bár a JSP alapvető célja az üzleti logika és a megjelenítés különválasztása, az A.5. ábrán látható példánk nem igazán demonstrálja ezt. Amennyire nem szerencsés HTML-kódot tenni a Java-forrásba, ugyanígy nem a legjobb választás Java-kódot elhelyezni HTML-oldalakba, akkor sem, ha ez még nyíltan nem sérti a sablon elvet.

Valójában az a helyzet, hogy a JSP bár nem kényszeríti rá a webalkalmazás készítőit a különböző logikai rétegek különválasztására,[5] eszközöket kínál arra, hogy ezt megtehessük. Ezeket az eszközöket a *JSP-akcióelemei* között találhatjuk meg.

A JSP-akcióelemek közül számunkra most az ún. *JavaBean-ek (babok)* kezelésére szolgáló elemek a legfontosabbak. A babok olyan Java-osztályok, amelyek bizonyos kódolási előírásoknak felelnek meg. Legfontosabb jellemzőjük, hogy rendelkeznek ún. *lekérdező* (angolul: *getter*) és *beállító* (angolul: *setter*) metódusokkal. Ezen metódusok a nevüknek

[5] Már csak azért sem, mert felfoghatjuk a JSP-t akár úgy is, mint szervletek írásának egy speciális, kényelmes módját. Lényegében tehát szervleteket írunk, így bármilyen „csúnyaságot” elkövethetünk.

megfelelően az egyes példányváltozók értékeit kérdezik le, illetve állítják be. A lekérdező és beállító metódusok neveit a példányváltozók neveiből származtatjuk. Például a `szam` példányváltozóhoz a `getSzam()` lekérdező metódusnév társul. Az A.6. ábrán láthatjuk az egyetlen számot tárolni képes babot.

```
package hu.cs.bme.szemweb;

public class SzamtaroloBab {
  private Integer szam;

  public Integer getSzam() {
    return(szam);
  }

  public void setSzam(Integer szam) {
    this.szam = szam;
  }
}
```

A.6. ábra. Egyetlen számot tárolni képes bab

A babok lehetnek egyszerű konténerek, amelyek bizonyos értékeket tárolnak például egy webes űrlap számára. A lekérdező és beállító metódusok tartalmazhatnak azonban olyan üzleti logikát is, amely szükséges a feladatuk elvégzéséhez. Például lehet, hogy egy beállító metódushívás hatására adatbázishoz kell fordulni, vagy valamilyen bonyolult számítás előzi meg az érték eltárolását. Alkalmazástól függően elképzelhető például, hogy elteszünk egy babba egy számot, de visszaolvasáskor már a négyzetét kapjuk meg stb. Megjegyezzük ugyanakkor, hogy sokszor szerencsésebb az ilyen jellegű „rejtett szolgáltatásokat” hanyagolni és úgy megírni a bab kódját, hogy az abban szereplő lekérdező és beállító metódusok működése átlátható legyen.

A babok kezelésére három JSP-akcióelem szolgál, ezek a következők:

- `jsp:useBean`
- `jsp:getProperty`
- `jsp:setProperty`

A `jsp:useBean` segítségével nevesíthetünk egy babot, amelyet a JSP-oldalon később használni szeretnénk.

```
<jsp:useBean id="bab" class="hu.cs.bme.hu.SzamtaroloBab" />
```

Alapesetben a `jsp:useBean` példányosítja a megadott osztályt, amelyre ezek után a megadott azonosítóval lehet hivatkozni. Nem történik tehát más, minthogy a webkonténer egy, az alábbihoz hasonló kódot generál és illeszt be a szervletbe:

```
SzamtaroloBab bab = new SzamtaroloBab();
```

Az igazán érdekes lehetőség az, hogy megadhatjuk babunk *hatáskörét*, és amennyiben a megadott hatáskörben ugyanilyen azonosítóval már létezik bab, akkor nem példányosítunk, hanem használjuk, amink van. Az alábbi babhatáskörök lehetségesek, amelyek megadják, hogy a babunknak hol kell láthatónak lennie (ennek biztosítása a webkiszolgáló feladata):

lap (page) Ez az alapértelmezett hatáskör, csak az adott lap érheti el a babot.

kérés (request) Az aktuális kérésben érvényes a bab, `request` objektumban tárolódik.

menet (session) Menetszinten érvényes a bab, a menetben tárolódik.

alkalmazás (application) Az összes olyan JSP-lap számára elérhető a bab, amelyek az adott szervlet kontextusában futnak.

Egy szervlet adott néven elhelyezheti a menetben például egy, az A.6. ábrán látható bab osztály példányát. Ilyenkor a `jsp:useBean` használatával elérhetjük, hogy JSP-oldalunkon felhasználhassuk ezt az objektumot. Ehhez az kell, hogy a bab osztályán kívül specifikáljuk a hatáskört is.

```
<jsp:useBean id="bab" class="hu.cs.bme.hu.SzamtaroloBab"
             scope="session" />
```

Ebben az esetben a webkiszolgáló megnézi, hogy létezik-e a menetben `bab` néven tárolt, adott osztályba tartozó példány, és ha igen, akkor a lap további részében `bab` néven ezt a példányt érjük el. Megtehetjük például, hogy egy szkriptelem segítségével lekérdezzük a babunk bizonyos mezőit.

```
<%= bab.getSzam() %>
```

```
<html>
  <head>
    <title>Eredmény JSP-lap</title>
    <meta http-equiv="content-type"
          content="text/html; charset=utf-8"/>
  </head>
  <body>
    <jsp:useBean id="eredmeny_bab" scope="session"
                 class="hu.cs.bme.szemweb.SzamtaroloBab" />
    <h1>
      Eredmény: <jsp:getProperty name="eredmeny_bab"
                                 property="szam" />
    </h1>
  </body>
</html>
```

A.7. ábra. Összeadás eredményének megjelenítése babból

A szkripteknél jobb megoldás, ha a `jsp:getProperty` és `jsp:setProperty` akciókat használjuk a getter, illetve a setter metódusok meghívásához. Az A.7. ábrán láthatunk egy

JSP-oldalt, amely a menetben előzetesen eltárolt babot felhasználva kiírja az összeadás eredményét.

Babok használatával elérhetjük, hogy a JSP-lapok közvetlenül csak és kizárólag a megjelenítéssel kapcsolatos információkat tárolják. Egy szervlet (amelyet akár egy másik JSP-lapban lévő űrlap segítségével érünk el) beállíthatja egy bab bizonyos tulajdonságait, például megadhatja a megfelelő számot, ami egy adott számítás végeredménye. Ezután eltárolhatja a babot valamilyen hatáskörben, majd továbbíthatja a vezérlést a megfelelő JSP-lapnak, amely a megfelelő akcióelemekkel eléri és felhasználja a babban tárolt értékeket.

```
package hu.cs.bme.szemweb;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import hu.cs.bme.szemweb.SzamtaroloBab;

public class OsszeadoMVC extends HttpServlet {
 public void doGet(HttpServletRequest request,
                   HttpServletResponse response)
                   throws ServletException, IOException {

  int sz1=Integer.parseInt(request.getParameter("szam1"));
  int sz2=Integer.parseInt(request.getParameter("szam2"));

  SzamtaroloBab bab = new SzamtaroloBab();
  bab.setSzam(new Integer(sz1+sz2));

  HttpSession session = request.getSession();
  session.setAttribute("eredmeny_bab",bab);

  RequestDispatcher dispatcher =
  getServletContext().getRequestDispatcher("gui.jsp");

  dispatcher.forward(request,response);
  }
}
```

A.8. ábra. Számösszeadós példánk szervlete MVC-szemléletben

Az elmondottakat az A.8. ábrán láthatjuk, ahol az eredeti számösszeadós szervletünket adtuk meg az újfajta szemléletben. Az eredmény tárolására egy babot használunk, a megjelenítést egy JSP-oldalra bízzuk. Érdemes összevetni a kódot az A.2. ábrán látható eredeti

megvalósítással. Igaz ugyan, hogy most három állományra van szükségünk (magára a szervletre, a bab osztály kódjára, valamint a JSP-lapra), de ez a látszólag pazarló felépítés nagy alkalmazások fejlesztése esetén bőségesen megtérül.

Példánkban azt is demonstráljuk, hogy hogyan tudjuk a vezérlést továbbítani egy szervletből, erre szolgál a `RequestDispatcher` osztály és annak a `forward` metódusa.

Minden igyekezetünk ellenére sokszor előfordul azonban, hogy szükséges bizonyos Java-kódrészletek elhelyezése a JSP-oldalunkban. A fejlesztőknek lehetőségük van arra, hogy ilyen esetben ne a JSP-szkriptelemeket használják, hanem ún. *egyedi akcióelemeket (tag-függvényeket)* hozzanak létre.

Egy-egy egyedi akcióelem elhelyezésekor az általunk írt Java-kód hívódik meg. Az egyedi akcióelemek írásával könyvünk nem foglalkozik, de megjegyezzük, hogy azokat általában könyvtárakba szervezzük, ahol több akcióelem kódját definiáljuk. Egy-egy ilyen könyvtárra a `taglib` JSP-direktívával hivatkozunk, ahogy az az alábbi JSP-lapon is látható:

```
<%@ taglib uri="/sajat_akcioelemek" prefix="szemweb" %>

<szemweb:tesztelFelhasznalot privilegium="0"/>

<html>
    ...
</html>
```

Példánkban a felhasználói jogosultságot ellenőrizzük egy egyedi akcióelemen keresztül. A megfelelő Java-kód eldönti, hogy az adott személy jogosult-e a honlap megtekintésére, és csak igenlő válasz esetén folytatódhat a JSP-lap kiszolgálása.