Budapesti Műszaki és Gazdaságtudományi Egyetem

Java technológia

Java Servlet és JavaServer Pages (JSP)

Java Web Services I.

- **Java Web Services** alatt webes szolgáltatások Java alapon történő megvalósítását támogató technológiák összességét értjük.
- A Java Web Serviceshez tartozó legfontosabb technológiák:
  - Java Servlet
  - JavaServer Pages (JSP)
  - JavaServer Faces (JSF)
  - JavaServer Pages Standard Tag Library (JSTL)
  - XML technológiák (JAXB, JAXP, JAXR, JAX-RPC)
  - SOAP with Attachments API for Java (SAAJ)

Java Web Services II.

- A felsorolt Java Web Services technológiák **API-kat** tartalmaznak.
- Ingyenes megvalósításuk a **Java Web Services Developer Packben (JWSDP)** található.
- Támogatott, vállalati szintű megvalósítások a **Sun ONE Studio**-ban, illetve a **Sun Java System Application Server**ben (régebben Sun ONE Application Server) találhatók.

WS-I I.

- A **Web Services Interoperability Organization (WS-I)** egy prominens ipari szereplők által létrehozott szervezet, melynek **célja a webes szolgáltatások közötti együttműködés lehetővé tétele** bevált technológiák (pl. XML, SOAP) segítségével.
- A WS-I nem szabványosító testület, a meglévő szabványokhoz dolgoznak ki ajánlásokat, útmutatókat az együttműködés elősegítése érdekében:
  - ☞ **Webes szolgáltatás profilok**: a webes szolgáltatásokkal kapcsolatos technológiák csoportosítása, illetve ajánlások együttes használatukhoz.
  - ☞ **Útmutatók** webes szolgáltatások implementálásához és teszteléséhez.

WS-I II.

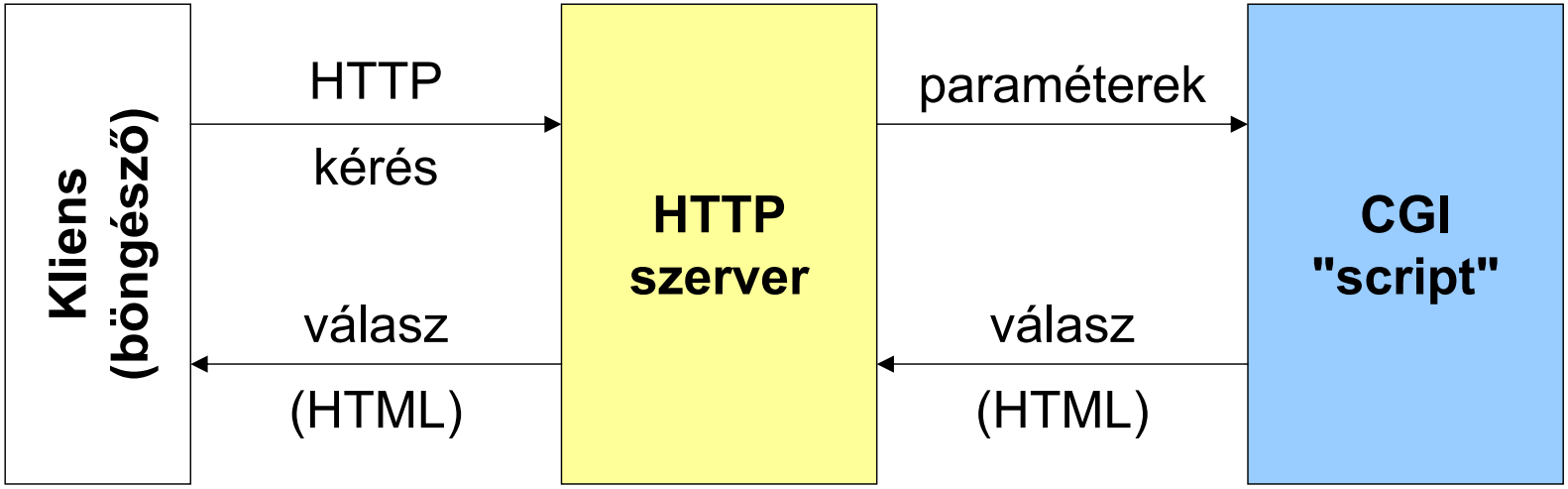
- Mivel a webes szolgáltatásokhoz kapcsolódó technológiák jelentős része nem teljesen kiforrott, a szabványok folyamatosan változnak.
- A WS-I ezért egyelőre csak egy alapvető profilt hozott létre, a **WS-I Basic Profile 1.0**-t.
- Ez felsorol néhány alapvető, webes szolgáltatást definiáló szabványt (pl. WSDL, SOAP, UDDI), és olyan kiegészítéseket tesz hozzájuk, amelyek ezen webes szolgáltatások együttműködését segítik.
- A JWSDP legújabb változata már teljesen konform a WS-I Basic Profile 1.0-val.

Webes alkalmazások megvalósítása I.

- A webes alkalmazások többnyire egy úgynevezett **háromszintű (three tier)** modellre épülnek.
- A három szintet a **kliens**, a **szerver**, és a **back-end** alkotja.
- A szerveren helyezkedik el az alkalmazás business logic része, a back-end pedig adattároló feladatokat lát el.
- A felhasználó **statikus és dinamikus tartalom** keverékét kapja, általában HTML formátumban (de lehet WML, XML, SVG, stb. is).
- A back-enddel most csak érintőlegesen foglalkozunk, célunk elsősorban a felhasználó felé közvetített tartalom előállítása (ezt néha front-endnek nevezik) a back-endből származó információ alapján.

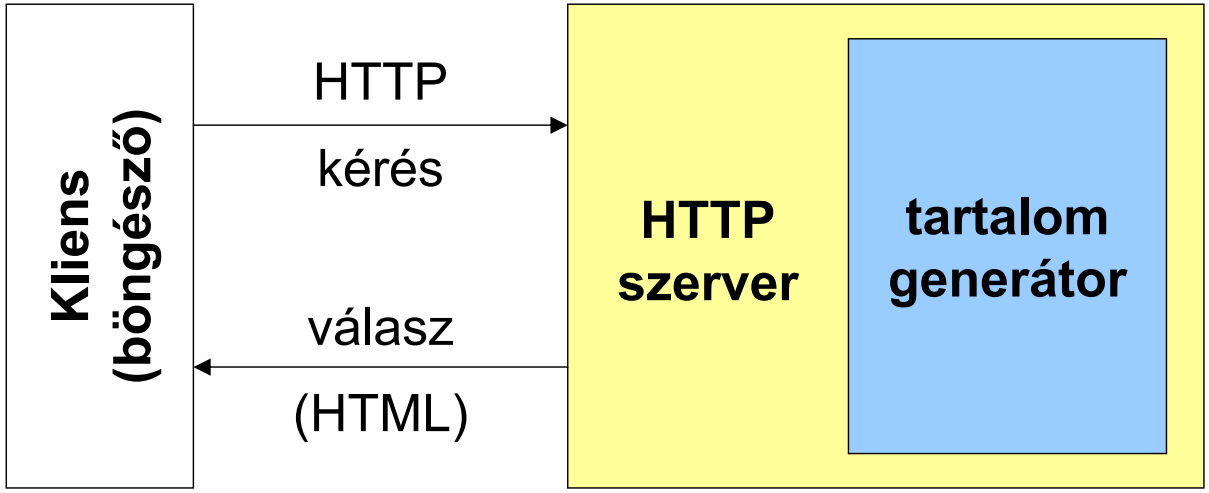
Webes alkalmazások megvalósítása II.

- Dinamikus tartalom megvalósításának "hagyományos" eszköze a **Common Gateway Interface (CGI)**.
- A CGI architektúra lehetővé teszi azt, hogy a HTTP szerverhez érkező igényt egy külső program dolgozza fel.



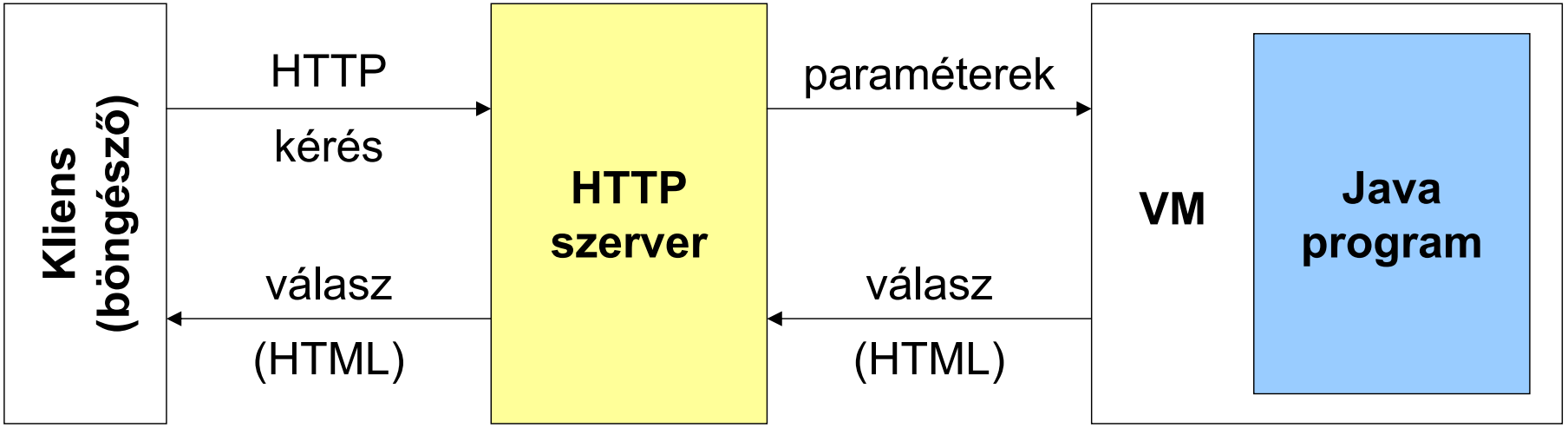
Webes alkalmazások megvalósítása III.

- Egyes technológiák esetén (pl. PHP) a dinamikus tartalmat létrehozó kód a **HTTP szerver processzen belül fut**.
- Ez a megoldás a külső processz indításánál lényegesen hatékonyabb, különösen nagy terhelés esetén.



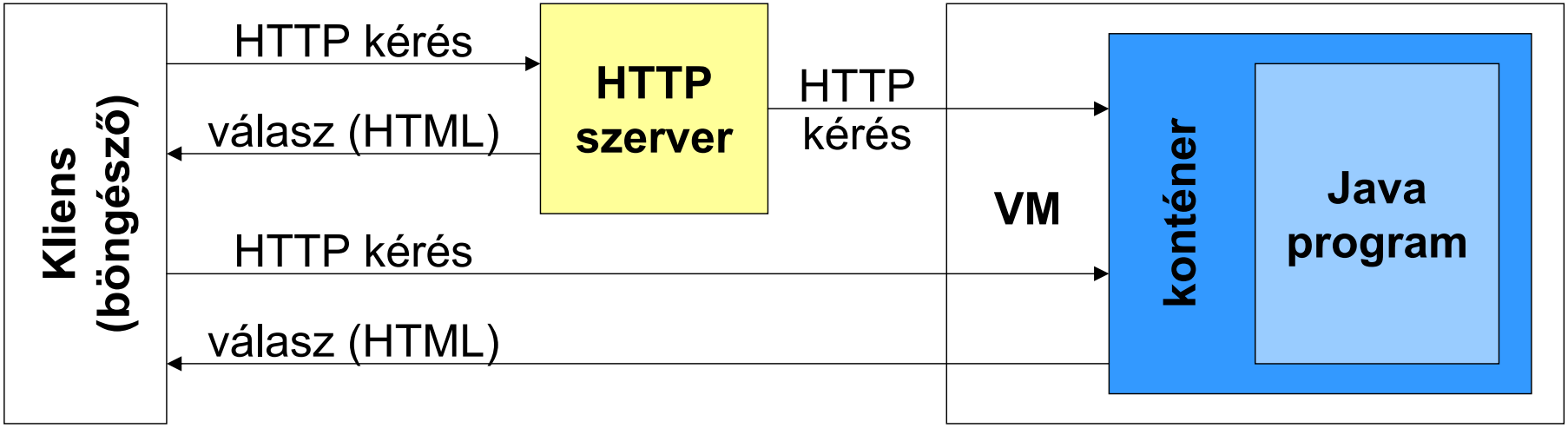
Webes alkalmazások megvalósítása IV.

- Java alkalmazása esetén a helyzet tovább bonyolódik, mert a Java programok futtatásához VM-re van szükség.
- A legprimitívebb megoldás, ha a Java programot CGI stílusban futtatjuk. A megoldás komoly hátránya, hogy a VM-et minden igény kiszolgálásakor újra el kell indítani, majd le kell állítani.



Webes alkalmazások megvalósítása V.

- Hatékonyabb megoldás, ha egy VM-ben egy ún. konténer alkalmazás fut folyamatosan, amely a HTTP szervertől is fogad igényeket, de akár önálló HTTP szerverként is képes működni.
- Tipikus eset, hogy a statikus oldalakat a HTTP szerver, a dinamikus tartalmat pedig a **konténer alkalmazás** szolgálja ki.

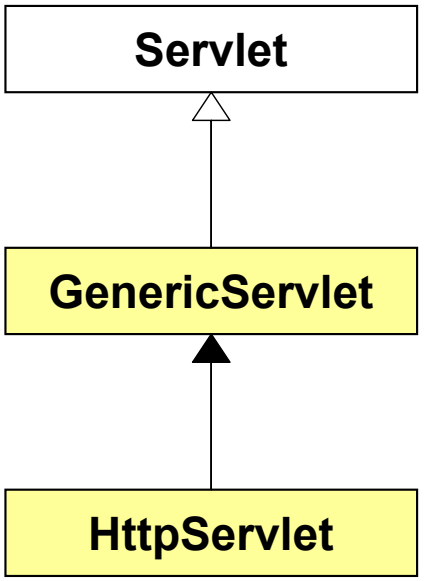


Servletek

- A Java Web Servicesben a webes alkalmazások megvalósításához használt szerver oldali Java programokat **servlet**eknek hívják. (server és servlet mint application és applet)
- A konténer alkalmazás neve **servlet konténer (servlet container)**.
- Ahogyan az applet sem képes önálló működésre, a servlet is csak egy servlet konténerben működik.
- A továbbiakban elmondottak a Java Servlet Specification 2.4-es, és a JavaServer Pages Specification 2.0-ás verziójára vonatkoznak.
- A felhasznált servlet konténer az Apache Software Foundation által fejlesztett Tomcat nevű szoftver, mely az 5.x verzió óta megvalósítja a fent említett specifikációkban foglaltakat.

Servletek a Javaban

- A servletekkel kapcsolatos interfészek és osztályok a javax.servlet és a `javax.servlet.http` csomagokban találhatók.
- A servletek a `Servlet` interfészből származó objektumok.
- A `GenericServlet` általános, protokollfüggetlen Servlet megvalósítások alapja.
- A `HttpServlet` a HTTP specifikumait is tartalmazza, egy szokványos webes alkalmazás készítésekor általában a `HttpServletből` kell leszármaztatnunk.



Servletek életciklusa I.

- A servleteket a servlet konténer **hozza létre**, **használja**, és **szünteti meg**.
- A servletek életciklusa az alábbi mozzanatokból áll:
  - betöltés és példányosítás
  - inicializálás
  - igénykiszolgálás
  - megszüntetés

Servletek életciklusa II.

- A servlet osztályt a konténer a szokványos Java osztálybetöltés segítségével **tölti be**, a servlet osztályának tehát elérhetőnek kell lennie.
- Betöltés után a konténer példányosítja a servlet osztályt.
- A konténer a Servlet interfész **init** metódusa segítségével **inicializálja** a servletet.
- A servletnek ekkor van lehetősége perzisztens konfiguráció betöltésére, és egyéb egyszeri műveletek (pl. adatbázis kapcsolat felépítés) elvégzésére.
- Az init metódus paramétereként a servlet kap egy ServletConfig interfészt implementáló objektumot.

ServletConfig és ServletContext

- A **ServletConfig** objektum az alábbi adatokat tartalmazza:
  - ☞ a servlet **nev**ét,
  - ☞ a webes alkalmazás deployment descriptorában (telepítés leíró, lásd később) definiált **inicializációs paraméterek**et,
  - ☞ egy **ServletContext objektumot**.
- A ServletContext a servlet futási környezetét definiálja, egy konténerhez egy példány tartozik (illetve elosztott konténer esetén VM-enként egy).
- A ServletContext a deployment descriptorban definiált, az alkalmazás egészére vonatkozó paramétereket, hozzáférési lehetőséget a servlet által elérhető erőforrásokhoz, loggolási lehetőséget, stb. tartalmaz.

Servletek életciklusa III.

- Inicializálás után (ha az sikeres volt) a servlet konténer felhasználhatja a servletet igények kiszolgálására.
- Előfordulhat az is, hogy a konténer soha nem használja az adott servlet példányt, hanem közvetlenül inicializálás után megszünteti.
- A konténer a Servlet interfész **service** metódusát hívja meg, ha a servlethez igény érkezett.
- A service metódus egy kérés és egy válasz objektumot kap, melyek a ServletRequest, illetve a ServletResponse interfészeket implementálják.
- HttpServlet esetén a kérés és válasz objektumok a HttpServletRequest és a HttpServletResponse interfészeket implementálják.

ServletRequest

- A **ServletRequest** objektum a beérkező igényről tartalmaz információt:
  - a kérés teljes törzsét,
  - a lokális és távoli címet és portot,
  - paramétereket,
  - protokollt,
  - attribútumokat.
- A HttpServletRequest a fentiek mellett HTTP fejléceket, a kérés típusát (GET, POST, stb.), session információt (lásd később), cookie-kat (lásd később) tartalmaz.

ServletResponse

- A **ServletResponse** objektum lényegében egy puffer, ahol a servlet válaszát elhelyezhetjük.
- Először megadhatjuk a tartalom típusát (content type).
- A getOutputStream és getWriter metódusok valamelyikével kapott OutputStream vagy Writer objektumokon keresztül bináris, illetve karakteres adattal tölthetjük fel a puffert.
- A HttpServletResponse a fentieken kívül lehetővé teszi HTTP hibakódok, cookie-k (lásd később), és HTTP fejlécek visszaküldését.

Servletek életciklusa IV.

- A servlet konténer általában **több konkurrensen érkező igényt is kiszolgálhat egy servlet példány segítségével**.
- Ez azt jelenti, hogy **a service metódust egyszerre több szálon is meghívhatja**, a servletet ennek figyelembevételével kell elkészíteni.
- Ha a servlet implementálja a SingleThreadModel interfészt, a servlet konténer sorbarendezi az igényeket, és nem hívja konkurrensen a service metódust, de ez az interfész a specifikáció legújabb verziójában már elavultnak minősül.
- A service metódust szabad szinkronizálni, de értelemszerűen rendkívüli mértékben rontja a teljesítményt.
- Egy-egy servlet osztályból több példány is létezhet ugyanabban a konténerben, tehát statikus mezők és metódusok használatakor erre figyelni kell.

Servletek életciklusa V.

- Ha már nincs szükség a servletre, a konténer **kivonhatja a forgalomból** a **destroy** metódus meghívásával.
- A destroy metódusban a servlet kimentheti az esetleges perzisztens konfigurációs adatokat, illetve felszabadíthatja a foglalt erőforrásokat (pl. adatbáziskapcsolatok lezárása).
- A forgalomból kivont servlet már nem inicializálható újra, új példányt kell létrehozni belőle.
- A destroy meghívása után a konténer elengedi a servletet, hogy a garbage collector begyűjthesse.

Példa: "Hello world"

```
package servlets;

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

public class TestServlet extends HttpServlet {

   public void service (ServletRequest request,ServletResponse response)
         throws IOException {
      PrintWriter out = new PrintWriter (response.getWriter ());
      out.println ("Hello world!");
   }
}
```

Webes alkalmazások felépítése I.

- Ahhoz, hogy a servletet használni tudjuk, telepítenünk kell a servlet konténerbe.
- Mivel a servlet konténerbe webes alkalmazások telepíthetők, a servletet be kell ágyaznunk egy alkalmazásba.
- A webes alkalmazás az alábbi elemekből áll:
  - servletek,
  - JSP oldalak,
  - segédosztályok,
  - statikus dokumentumok (HTML, képek, hangok, stb.),
  - kliens oldali appletek, beanek, osztályok, stb.,
  - az egészet összefogó metainformáció.

Webes alkalmazások felépítése II.

- A webes alkalmazás könyvtárhierarchiájában mindig kell legyen egy WEB-INF nevű könyvtár, amely az alábbiakat tartalmazza:
  - ☞ WEB-INF/web.xml: a deployment descriptor, vagy telepítés leíró,
  - ☞ WEB-INF/classes: a servletek osztályai és kisegítő osztályok,
  - ☞ WEB-INF/lib/*.jar: tetszőleges, ide telepített JAR file tartalma elérhető az alkalmazás osztályai számára.
- A konténer osztálybetöltője először a WEB-INF/classes könyvtárból, aztán a WEB-INF/lib-ben lévő JAR-okból tölti be az osztályokat.
- A WEB-INF könyvtár nem tartozik az alkalmazás publikus könyvtáraihoz, ezért az itt található file-okat a konténer nem szolgálja ki közvetlenül a klienseknek.

Webes alkalmazások telepítése I.

- A webes alkalmazásokat a servlet konténer által meghatározott könyvtárhierarchiában kell elhelyezni.
- Ez konténerenként változhat, a Tomcatben **virtuális hoszt**okat definiálhatunk, melyek egy-egy külön könyvtárban tárolják az alkalmazásokat, ezek neve általában **webapps**.
- A konténerben elhelyezett alkalmazásokat a konténer indításkor, vagy futás közben manuálisan vagy automatikusan deríti fel, és telepíti (deployment).

Webes alkalmazások telepítése II.

- Mivel nem feltétlenül kényelmes az alkalmazásokat file-onként mozgatni, a JAR file-okhoz hasonlóan a webes alkalmazásokat is összepakolhatjuk egy file-ba, ezek a **Web ARchive (WAR)** file-ok.
- Egy WAR lényegében egy JAR file, a neve egyszerűen arra utal, hogy nem egy Java alkalmazásról, vagy appletről van szó, hanem egy webes alkalmazásról.
- A konténerek támogatják a WAR file-ból való telepítést is, ilyenkor nincs más dolgunk, mint a WAR file-t elhelyezni az erre kijelölt helyre, a konténer automatikusan kitömöríti, és az alkalmazást telepíti.

URL leképezés I.

- Mivel a servlet konténer lényegében web szerverként viselkedik, a kérésekben előforduló URL-eket le kell képezni a tényleges kiszolgálást végző servletekre.
- A leképezés két fő mozzanatból áll:
  - ☞ Annak megállapítása, hogy az adott URL a telepített alkalmazások közül melyikre vonatkozik.
  - ☞ Annak megállapítása, hogy az alkalmazáson belül melyik servlet szolgálja ki a kérést, esetleg statikus tartalomra vonatkozik-e.
- Az első kérdés megválaszolásához meg kell tudni különböztetni az egyes alkalmazásokat.

URL leképezés II.

- Minden alkalmazáshoz tartozik egy csak rá jellemző **context path** (kontextus útvonal).
- Ezt a context path-t az alkalmazás telepítésekor kell megadnunk. Ha explicite nem adjuk meg a context path-t, a virtuális hoszt könyvtára alatti könyvtár nevével fog megegyezni.
- Ha például az alkalmazást a webapps/myapp könyvtárba telepítettük, ahol a webapps a virtuális hoszt könyvtára, és más context path-t nem adtunk meg, akkor ennek értéke /myapp lesz.
- Az alkalmazás kiválasztásához a kérésben szereplő URL prefixét kell illeszteni a lehetséges context path-okra. A kérés a leghosszabban illeszkedő context path-hoz tartozó alkalmazáshoz kerül.

URL leképezés III.

- Az alkalmazáson belül a megfelelő servlet kiválasztásához a deployment descriptorban megadott leképezéseket használja fel a konténer, melyek URL mintákhoz rendelnek servleteket.
- Ha az adott mintára illeszkedik az URL megfelelő része, a kérés mintához tartozó servlethez kerül.
- Kérdés, hogy melyik az URL "megfelelő" része?

URL leképezés IV.

- Tegyük fel, hogy a kérésben az alábbi URL szerepelt:
  (az általánosság kedvéért HTTP GET kérés)

```
http://host.domain.com/applications/servlets/util/MyServlet?p=12&q=9
```

- A http neve **séma** (URI schema).
- A host.domain.com a számítógépet jelöli ki.
- Tegyük fel, hogy az első leképezési lépésben a leghosszabban egyező context path `/applications/servlets` volt. Ez kijelöli a webes alkalmazást.
- A ? és az azt követő rész az ún. **path parameter** (útvonal paraméter), amely HTTP GET kérésben paraméterek átadására szolgál.
- Az alkalmazáson belül a servletet tehát a /util/MyServlet rész jelöli ki.

URL leképezés V.

- A deployment descriptorban az URL minta szerint négy típusú leképezést adhatunk meg:
  - /-el kezdődő, és /*-gal végződő minták: az adott útvonalra illeszkedő összes URL kijelölése,
  - *.-tal kezdődő minták: az adott kiterjesztésű névvel végződő URL-ek kijelölése,
  - az egyetlen /-ből álló minta: az alapértelmezett servletet jelöli ki, arra az esetre, ha egyik másik minta sem illeszkedik,
  - minden egyéb minta: a mintával pontosan megegyező URL-eket jelöli ki.
- Fontos, hogy a kiterjesztésekre vonatkozó minták nagyobb precedenciájúak, mint az útvonalakra vonatkozó minták.

URL leképezés VI.

- Legyenek megadva az alábbi leképezések:

```
/util/special/*             SpecialServlet
/util/*                     UtilServlet
/myservlet                  MyServlet
*.zaz                       ZazServlet
/                           DefaultServlet
```

- Néhány példa útvonalakra, és a leképezés eredményeképpen adódó servletekre:

```
/util/special/valami.txt    SpecialServlet
/util/special/valami.zaz    ZazServlet              // kiterjesztés
/myservlet                  MyServlet
/myservlet/valami.txt       DefaultServlet  // nem pontos illeszkedés
/myservlet/valami.zaz       ZazServlet              // kiterjesztés
/valami/egyeb               DefaultServlet
                            DefaultServlet          // üres útvonal
```

A deployment descriptor I.

- A deployment descriptor az alábbi információkat tartalmazza:
  - a ServletContext inicializációs paraméterei,
  - session-ök konfigurációja,
  - servlet/JSP definíciók,
  - URL leképezések,
  - MIME típus leképezések,
  - welcome file-ok listája,
  - hiba oldal definíciók,
  - biztonsági beállítások.

A deployment descriptor II.

- Ahhoz, hogy a példában szereplő servletet kipróbálhassuk, az alábbi deployment descriptorra van szükségünk:

```
<web-app>
   <servlet>                                <!-- Servletek definíciója -->
      <servlet-name>TestServlet</servlet-name>
      <servlet-class>servlets.TestServlet</servlet-class>
   </servlet>

   <servlet-mapping>              <!-- Servletek leképezése URL-ekre -->
      <servlet-name>TestServlet</servlet-name>
      <url-pattern>/</url-pattern>              <!-- Default servlet -->
   </servlet-mapping>
</web-app>
```

A deployment descriptor III.

- A fenti deployment descriptor az alábbiakat határozza meg:
  - ☞ A TestServlet nevű servlet kódját a servlet.TestServlet osztály tartalmazza.
  - ☞ Minden, az adott webes alkalmazáshoz érkező kérést a TestServlet szolgál ki.

A "Hello world" servlet telepítése

- A kipróbáláshoz szükséges webes alkalmazást az alábbi módon helyezzük el a filerendszerben:

```
/webapps
   /test
      /WEB-INF
         web.xml
         /classes
            TestServlet.class
```

- A servletet az alábbi URL-lel serkenthetjük működésre:

```
http://localhost:8080/test
```

- ..., ahol 8080 a servlet konténer portja (Tomcat default érték).
- Az URL leképezés tulajdonságai miatt akár az alábbi URL is megfelel:

```
http://localhost:8080/test/valami/akarmi.txt
```

Paraméterek

- Servleteknek kétféle paramétert adhatunk át:
  - **Inicializációs paraméterek**: a deployment descriptorban adhatjuk meg, tipikusan a telepítési környezettel kapcsolatos információkat tartalmazzák.
  - **Kérések paraméterei**: a kérésekben szereplő, a kliens által küldött paraméterek.

Inicializációs paraméterek I.

- Egészítsük ki a deployment descriptort egy inicializációs paraméterrel:

```
<web-app>
   <servlet>
      <servlet-name>TestServlet</servlet-name>
      <servlet-class>servlets.TestServlet</servlet-class>
      <init-param>
         <param-name>smtp-server</param-name>
         <param-value>smtp.valami.hu</param-value>
      </init-param>
   </servlet>

   <servlet-mapping>
      <servlet-name>TestServlet</servlet-name>
      <url-pattern>/</url-pattern>
   </servlet-mapping>
</web-app>
```

Inicializációs paraméterek II.

- Egészítsük ki a servlet kódját is a paraméter kiolvasásával:

```
public class TestServlet extends HttpServlet {

   public void init (ServletConfig config) {
      smtpServerAddress = config.getInitParameter ("smtp-server"));
   }

   public void service (ServletRequest request,ServletResponse response)
         throws IOException {
      PrintWriter out = new PrintWriter (response.getWriter ());
      out.println ("Hello world!<br/>");
      out.println ("Az SMTP szerver címe: " + smtpServerAddress);
   }

   private String smtpServerAddress;
}
```

Inicializációs paraméterek III.

- Az inicializációs paramétereket a `ServletConfig` objektum tartalmazza.
- A paramétereket célszerűen az init metódusban olvassuk ki, a `ServletConfig` objektum `getInitParameter`, illetve `getInitParameterNames` metódusok segítségével.

Kérések paraméterei I.

- A kliens a kérésekben továbbíthat paramétereket a servletnek, ezeket a service metódus által megkapott ServletRequest objektum tartalmazza.
- A paraméterekhez a `ServletRequest` objektum `getParameter`, `getParameterMap`, `getParameterNames` és `getParameterValues` metódusai segítségével férhetünk hozzá.
- A paraméterek protokollfüggő módon tárolódnak a kérésben, a ServletRequest objektum ezt többnyire elfedi.
- HTTP protokoll esetén a HttpServletRequest objektum megfelelő metódusai segítségével (elvileg) akkor kaphatók meg a paraméterek, ha HTTP POST metódussal történt a paraméterátadás.
- Noha a specifikáció szerint a HTTP GET kérésekben szereplő paramétereket nem lehet így elérni, a Tomcatben ez lehetséges.

Kérések paraméterei II.

- Dolgozzuk fel az alábbi HTML oldalban lévő form paramétereit egy servlettel:

```
<html>
   <head>
      <title>HTTP POST</title>
   </head>
   <body>
      <form action="/test" method="post">
         Alma: <input type="text" name="alma"/><br/>
         Körte: <input type="text" name="korte"/><br/>
         <input type="submit" value="Mehet"/>
      </form>
   </body>
</html>
```

Kérések paraméterei III.

- Módosítsuk a servlet kódját úgy, hogy a form paramétereit kilistázza:

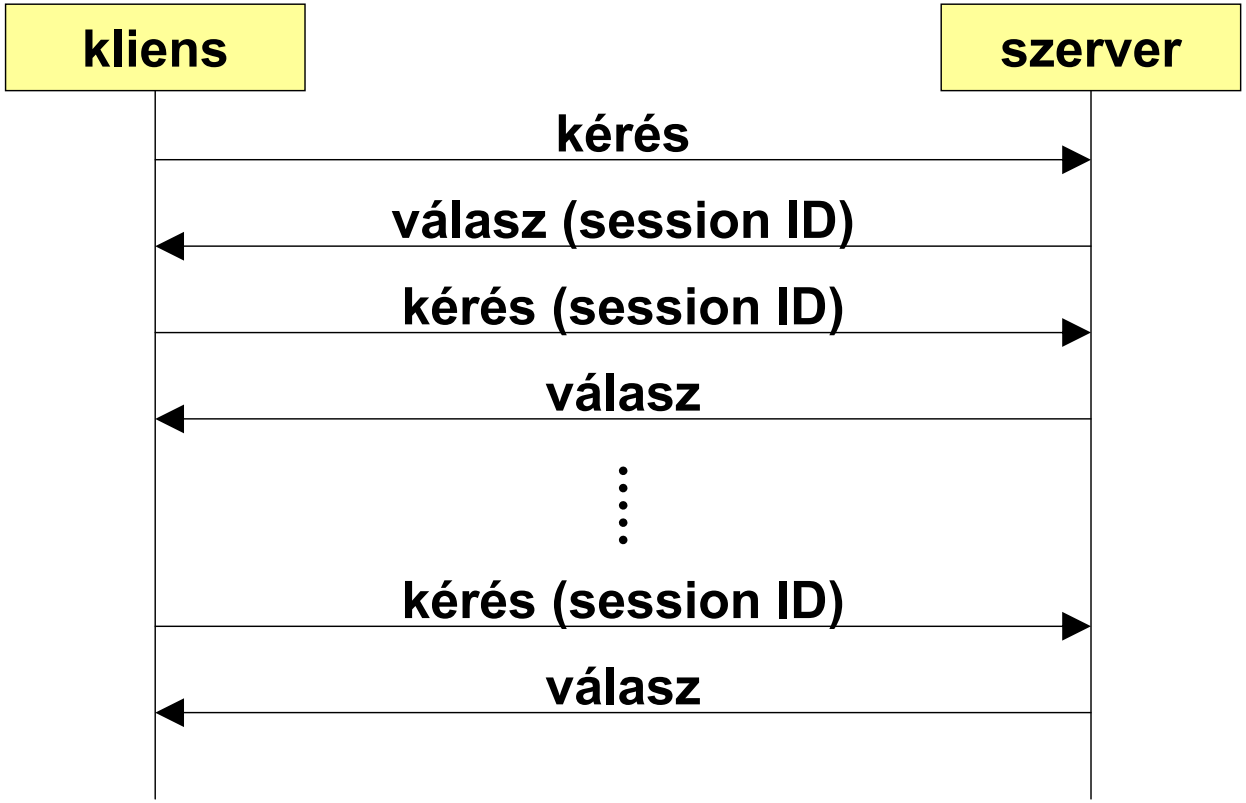
```
public class TestServlet extends HttpServlet {

   public void service (ServletRequest request,ServletResponse response)
         throws IOException {
      PrintWriter out = new PrintWriter (response.getWriter ());
      Enumeration e = request.getParameterNames ();
      while (e.hasMoreElements ()) {
         String name = (String) e.nextElement ();
         out.println (name + "=" + request.getParameter (name) + "<br/>");
      }
   }
}
```

Sessionkezelés I.

- A HTTP **állapotmentes** (stateless) protokoll.
- Gyakran szükség van arra, hogy valamilyen módszerrel összekapcsoljuk kérések és válaszok egy sorozatát.
- Az ilyen sorozatokat nevezzük **session**nek.
- A sessionök néhány lehetséges célja:
  - **visszatérő felhasználók azonosítása**, például honlapok személyre szabása céljából,
  - **jelszavas védelem** megvalósítása,
  - műveletek **tranzakciók**ba szervezése, stb.

Sessionkezelés II.

- A HTTP-ben a sessionkezelés elve a következő ábrán látható.



Sessionkezelés III.

- A **session azonosítók**at át kell vinni a szerverről a kliensre, majd a további kérések során a kliensről a szerverre.
- Az azonosítók továbbítására három módszer valamelyikét használják a servletek:
  - cookie-k,
  - SSL sessionök,
  - URL rewriting.
- HTTPS esetén az SSL sessionöket közvetlenül fel lehet használni HTTP sessionök azonosítására.
- Az URL rewriting akkor szükséges, ha a kliens nem támogatja (vagy nem engedélyezi) cookie-k használatát.

Sessionkezelés IV.

- A servlet fejlesztő számára a sessionök kezelése teljesen transzparens, a három módszer közül a konténer automatikusan választ.
- Egy sessiont egy `HttpSession` objektum reprezentál, melyet a `HttpServletRequest` objektum `getSession` metódusával kérhetünk el.
- A `HttpSession` objektumban nevekhez köthetünk objektumokat, ezeket a session attribútumainak hívjuk.
- Az adott sessionhöz tartozó, soron következő kérések `HttpServletRequest` objektumaitól megkapható `HttpSession` objektum tartalmazza a session korábban beállított attribútumait.
- Az attribútumok segítségével tetszőleges információt "átörökíthetünk" egy session kérései között.

Sessionkezelés V.

- A HttpSession objektum isNew metódusával lekérdezhetjük, hogy **a session új-e**. A session akkor új, ha:
  - ☞ a kliens még nem tud róla (vagyis a konténer most hozta létre),
  - ☞ a kliens nem "harap rá", vagyis az új kérésben nem küldte vissza a session azonosítót.
- Amíg nem kapunk egy olyan HttpSession objektumot, amely nem minősül újnak, nem tételezhetjük fel, hogy a következő kérés a session része lesz.
- Mivel a HTTP állapotmentes protokoll, nincs definiálva a kapcsolat bontása, ezáltal a sessionök vége sem.
- A sessionök egy beállítható hosszúságú **inaktivitási időszak végén szűnnek meg**.

Példa: Sessionkezelés

```
public class TestServlet extends HttpServlet {
   public void service (ServletRequest request,ServletResponse response)
         throws IOException {
      PrintWriter out = new PrintWriter (response.getWriter ());
      HttpSession session = ((HttpServletRequest) request).getSession ();
      if (session.isNew ())
         out.println ("<form action=\"/testServlet\" method=\"post\">" +
               "<input type=\"text\" name=\"valami\">" +
               "<input type=\"submit\" value=\"Mehet\"></form>");
      else {
         String param = request.getParameter ("valami");
         if (param != null) {
            session.setAttribute ("valami",param);
            out.println ("Uj ertek: " + param);
         } else
            out.println ("Regi ertek: " + session.getAttribute ("valami"));
      }
   }
}
```

JavaServer Pages (JSP) I.

- A **JavaServer Pages (JSP)** technológia dinamikus web tartalom létrehozására szolgál.
- A JSP segítségével szövegesen írhatjuk le, hogyan kell kérésekből dinamikus válaszokat létrehozni.
- A JSP három legfontosabb elvi alkotóeleme:
  - ☞ **statikus** (template) **szöveg**, amely az oldalak legnagyobb, fix részét alkotja,
  - ☞ **dinamikus tartalom** hozzáadása,
  - ☞ a **funkcionalitás leválasztása** a megjelenítésről JavaBeans komponensek és JSP tag library-k segítségével.

JavaServer Pages (JSP) II.

- A fenti elemek révén
  - ☞ a JSP oldalak **platformfüggetlenek**, mert az elkészült JSP oldalak és a hozzájuk tartozó komponensek bármely JSP kompatíbilis szerverre telepíthetők,
  - ☞ a **fejlesztő és tartalomkészítő szerepek függetlenek**, mert a funkcionalitást biztosító kódrészletek az oldalból kiemelhetők,
  - ☞ az elkészült komponensek és tag library-k **újra felhasználhatók**,
  - ☞ a **statikus és dinamikus tartalom jól elkülöníthető**.

JavaServer Pages (JSP) III.

- A JSP technológiát jellemzően HTML oldalak generálására használják.
- Minden JSP oldalnak létezik ekvivalens **XML nézete**, így a technológia XML dokumentumok generálására is alkalmas.
- A JSP lehetővé teszi más JSP oldalak beágyazását is, így nem csak egész JSP oldalak, hanem JSP szegmensek (korábban fragmensek) is létezhetnek.
- A JSP file-ok ajánlott kiterjesztései:
  - JSP oldalak: `jsp`
  - JSP szegmensek: `jspf`
  - XML formátumú JSP dokumentumok: `jspx`

JSP oldalak életciklusa

- A JSP oldalak igények kiszolgálása előtt **servletté fordulnak le**, így életciklusuk és többszálú viselkedésük a servletekéhez hasonló. A JSP oldalak fordítására képes servlet konténereket **JSP konténer**eknek nevezzük.
- A JSP oldalak életciklusa két fő részből áll, ezek a **fordítási fázis** és a **végrehajtási fázis**.
- A JSP oldalakat a webes alkalmazás telepítésétől kezdve az első igény kiszolgálásáig bármikor lefordíthatja a konténer.
- A fordítás jellemzően közvetlenül a JSP oldalhoz érkező első igény kiszolgálása előtt történik.
- Mivel ez a módszer az oldal első letöltésekor érezhető késést eredményez, a JSP oldal az alkalmazás telepítése előtt is lefordítható.

JSP oldalak elemei

- A JSP oldalak az alábbi elemekből állnak:
  - statikus (template) szöveg,
  - direktívák,
  - (expression language elemek),
  - script elemek:
    - deklarációk,
    - scriptletek,
    - kifejezések,
  - akciók.

Direktívák

- A direktívák a JSP konténernek szólnak, és az oldal fordítását és futtatását szabályozzák.
- A direktívák szintakszisa az alábbi:

```
<%@ direktíva attribútum="érték" ... %>
```

- Három direktíva létezik:
  - **page**: az oldal bizonyos tulajdonságainak beállítására szolgál,
  - **taglib**: a JSP tag library-kkal kapcsolatos, lásd később,
  - **include**: szöveg beillesztésére szolgál.

A page direktíva I.

- A page direktíva attribútumai közül a fontosabbak:
  - **language**: Az oldalba ágyazott deklarációk, scriptletek és kifejezések nyelvét határozza meg. A JSP 2.0 egyedül a java értéket definiálja. Ez azt jelenti, hogy az oldalba ágyazott kódrészleteknek meg kell felelniük a Java Language Specification-nek.
  - **import**: A Java programok elején található import deklarációkhoz hasonló funkciójú, az oldalban egyszerű névvel hivatkozható típusokat definiálja. Egy import attribútum lehet, értéke egy vesszővel elválasztott lista. Az alapértelmezett lista:
    "`java.lang.*,javax.servlet.*,javax.servlet.jsp.*,javax.servlet.http.*`"

A page direktíva II.

- **session**: Ha true-ra állítjuk (ami a default érték), a létrejövő servlet automatikusan létrehoz egy HTTP sessiont, ami az oldalba ágyazott kódból egy implicit változón (lásd később) elérhető.
- **isErrorPage**: Ha értéke true (a default false), ez az oldal egy másik JSP oldal hibaoldalaként (lásd errorPage attribútum) funkcionál. Ebben az esetben az oldalban létrejön egy exception implicit változó, amely a lekezelendő hibát tartalmazza.
- **errorPage**: Ha az oldal végrehajtása során kivétel dobódik, a kivétel az itt megadott URL-re továbbítja. Ha az URL egy másik JSP oldalra mutat, a másik oldal isErrorPage attribútuma true értékű kell legyen. A kivétel objektum az aktuális ServletRequest objektumban attribútumként tárolódik, majd a kérés továbbadódik a megadott URL-re. Az errorPage értéke felülbírálja a deployment descriptorban megadott hibaoldalakat.

A page direktíva III.

- **contentType**: A JSP lap eredményének MIME típusát és kódolását (encoding) adja meg. Formátuma "TYPE", vagy "TYPE;charset=CHARSET", ahol a TYPE egy MIME típus, a CHARSET pedig egy kódolás IANA szerint érvényes neve. A default érték "text/html", ha a JSP oldal HTML szintakszisú, és "text/xml", ha XML szintakszisú. A CHARSET default értéke a pageEncoding attribútumhoz hasonlóan határozható meg.
- **pageEncoding**: A JSP oldal kódolását adja meg, az érték egy kódolás IANA szerint érvényes neve kell legyen. A default érték HTML szintakszisú oldal esetén a contentType-ban definiált érték, ha meg van adva, ha nem, akkor "ISO-8859-1" (vagyis Latin-1), XML szintakszisú oldal esetén, ha a dokumentum nem definiálja, "UTF-8".

Az include direktíva

- Az include direktíva szöveg, vagy kód **fordításidejű** beillesztésére szolgál.
- Szintakszisa:

```
<%@ include file="relatívURL" %>
```

- A megadott URL-en található file tartalma bemásolódik a direktíva helyére.

Script elemek

- A script elemek segítségével az oldalon használt script nyelven írt kódrészletek (a továbbiakban Java kódrészletek) közvetlenül beágyazhatók az oldalba.
- A script elemek három fajtája létezik:
  - **deklarációk**: változók és metódusok deklarálása,
  - **scriptletek**: végrehajtandó kódrészletek,
  - **kifejezések**: kiértékelendő kifejezések.

Deklarációk

- A deklarációk változók és metódusok deklarálását teszik lehetővé.
- Egy deklarációban egy vagy több deklaráció helyezkedhet el.
- Szintakszis:

```
<%! deklarációk %>
```

- A deklarációk nem hoznak létre kimenetet az oldalra adott válaszban.
- Figyelni kell arra, hogy a **deklarációk globálisak az oldalra nézve**, és az oldalt **egyszerre több szálon is végrehajthatják**. Az alábbi deklaráció éppen ezért veszélyeket rejt magában:

```
<%! int i = 0; %>
```

- Metódus deklarációja

```
<%! public int getTwo () { return 2; } %>
```

Scriptletek I.

- A scriptletek tetszőleges kódrészletet tartalmazhatnak.
- Szintakszis:

```
<% tetszőleges kód %>
```

- A scriptletek létrehozhatnak kimenetet az oldal válaszában, és módosíthatnak számukra látható objektumokat és változókat.
- Az oldalban előforduló scriptleteket az oldal fordítása során a JSP konténer összekombinálja, így **együttesük egy érvényes utasítássorozatot kell adjon**.
- A fentiek értelmében az oldal statikus részei a scriptletek által meghatározott vezérlési szerkezetek hatása alá kerülnek.

Scriptletek II.

- Példák:

```
<%! int i = 0; %>
...
<% i++; %>
```

```
<% if (loveClient ()) { %>
   Szép jó napot, magasságos Uram!
<% } else { %>
   Elmész te a bánatba!
<% } %>
```

```
<% for (int i = 0;i < 10;i++) { %>
   Ezt tízszer írom ki.
<% } %>
```

Kifejezések

- A JSP kifejezések egy Java kifejezést tartalmaznak, amelynek értéke kiértékelés után bekerül az oldal válaszába.
- A kifejezéseknek lehet mellékhatásuk.
- A kifejezések az oldalon, illetve attribútumértékeken belül balról jobbra értékelődnek ki.
- Szintakszis

```
<%= kifejezés %>
```

- Példák:

```
<%= new java.util.Date () %>
```

```
<% for (int i = 0;i < 10;i++) { %> i=<%= i %></br> <% } %>
```

Kitérő: implicit objektumok I.

- A JSP oldalak script elemeiben néhány objektum mindig elérhető, ezeket **implicit objektum**oknak nevezzük.
- Az implicit objektumoknak többféle érvényességi körük (scope) lehet:
  - **page**: a JSP oldal, a PageContextben tárolódnak,
  - **request**: azok a JSP oldalak, amelyek az adott kérés kiszolgálásában vesznek részt, a ServletRequestben tárolódnak,
  - **session**: az adott sessionhöz tartozó kérések kiszolgálásában résztvevő JSP oldalak, a HttpSessionben tárolódnak,
  - **application**: az adott alkalmazáshoz tartozó JSP oldalak, a ServletContextben tárolódnak.

Kitérő: implicit objektumok II.

Név	Típus	Érvényesség	Magyarázat
request	javax.servlet.(http.Http)ServletRequest	request	A JSP oldal meghívását eredményező kérés.
response	javax.servlet.(http.Http)ServletResponse	page	A kérésre adott válasz.
pageContext	javax.servlet.jsp.PageContext	page	A JSP oldalhoz tartozó PageContext objektum.
session	javax.servlet.http.HttpSession	session	A sessionhöz tartozó HttpSession objektum, csak HTTP protokoll esetén.
application	javax.servlet.ServletContext	application	A JSP oldalból létrejött servlethez tartozó ServletContext objektum.

Kitérő: implicit objektumok III.

- Megjegyzés: a jsp, _jsp és _jspx prefixű (kis- és nagybetűktől függetlenül) objektumneveket a JSP specifikáció foglaltnak tekinti.

Név	Típus	Érvényesség	Magyarázat
out	javax.servlet.jsp.JspWriter	page	Az oldal kimenetéhez tartozó Writer objektum.
config	javax.servlet.ServletConfig	page	A JSP oldalból létrejött servlethez tartozó ServletConfig objektum.
page	java.lang.Object	page	A JSP oldalt megvalósító objektum (Java esetén megegyezik a this referenciával).
exception	java.lang.Throwable	page	Hibaoldal esetén a kapott kivétel objektum.

Akciók

- Az akciók kimenetet generálhatnak az oldal válaszába, illetve létrehozhatnak és módosíthatnak objektumokat.
- Az akciók XML tagekként jelennek meg, ennek megfelelően lehetnek üresek és nem üresek.
- Az akciók attribútumai lehetnek konstansok, vagy scriptlet kifejezések, de a kettőt nem lehet keverni:

```
<akció attr="érték1"/>                                    // OK
<akció attr="<%= i+2 %>"/>                                // OK
<akció attr="i=<%= i %>"/>                              // Hibás
```

- Akciókat a felhasználó is készíthet a Tag Extensions használatával (lásd később), és léteznek szabványos akciók is.

Szabványos akciók

- A szabványos akciókat minden JSP konténer támogatja.
- A szabványos akciók külön XML namespace-ben vannak, a default prefix jsp.
- A következőkben a fontosabb szabványos akciókat tekintjük át:
  - <jsp:useBean>
  - <jsp:setProperty> és <jsp:getProperty>
  - <jsp:include>
  - <jsp:forward>
  - <jsp:param>
  - <jsp:element>, <jsp:attribute> és <jsp:body>

Szabványos akciók: <jsp:useBean> I.

- Egy megadott érvényességi körben létező Java objektumot hozzárendel egy script változóhoz, vagy a megadott érvényességi körben új példányt hoz létre.
- Attribútumai:
  - **id**: Az objektum azonosítója a megadott érvényességi körben, valamint a script változó neve.
  - **scope**: Az érvényességi kör. Lehet page, request, session, vagy application.
  - **class**: Az objektum osztályneve.
  - **beanName**: Egy Java Bean neve (lásd később), a class helyett használható.
  - **type**: A script változó típusa, ha nem egyezik meg az osztállyal.

Szabványos akciók: <jsp:useBean> II.

- Az akció törzse lehet üres vagy nem üres, ha nem üres, a tartalma akkor hajtódik végre, ha új példány jött létre a megadott osztályból (vagyis ha a megadott érvényességi körben a megadott azonosítóval nem létezett példány).
- Példa:

```
<jsp:useBean id="alma" scope="session" class="tools.Gyumolcs">
   Ez az uj gyumolcs: <%= alma %>
</jsp:useBean>
```

Szabványos akciók: <jsp:setProperty> I.

- Egy Java Bean tulajdonságait állítja be három lehetséges módon:
  - Egy adott tulajdonság beállítása konkrét értékkel.
  - Egy adott tulajdonság beállítása a kérésből származó paraméter értéke alapján.
  - A kérésben szereplő paraméterek neveivel megegyező nevű tulajdonságok beállítása a megfelelő paraméterértékekkel.
- Attribútumai:
  - **name**: Egy Java Bean példány neve (a példányra mutató script változó neve).
  - **property**: A beállítandó tulajdonság neve, ha üres, a kérés összes paraméterét állítja be.

Szabványos akciók: <jsp:setProperty> II.

- További attribútumai:
  - ☞ **param**: A property attribútumban megadott tulajdonságot beállítja a megadott kérés paraméter értékére. Ha a paraméternek nincs értéke, az akció hatástalan.
  - ☞ **value**: A property attribútumban megadott tulajdonságot beállítja a megadott értékre.
- A param és a value attribútum egyszerre nem szerepelhet.
- Példa:

```
<jsp:setProperty name="alma" property="szin" param="alma-szin"/>
```

- A fenti példa az alma változóban lévő Java Bean szin tulajdonságát állítja be a kérésben szereplő alma-szin nevű paraméter értékére.

Szabványos akciók: <jsp:getProperty>

- Egy Java Bean egy tulajdonságát kiírja a JSP oldal válaszába (az out implicit objektum által definiált Writerre).
- Attribútumai:
  - **name**: Egy Java Bean példány neve (a példányra mutató script változó neve).
  - **property**: A tulajdonság neve.
- Példa:

```
A gyumolcs szine: <jsp:getProperty name="alma" property="szin"/>
```

Szabványos akciók: <jsp:include>

- Statikus vagy dinamikus tartalom beemelése.
- A megadott tartalom közvetlenül az oldal válaszába kerül további feldolgozás nélkül.
- Nem összetévesztendő az include direktívával, mert az fordításidőben hajtódik végre, az include akció pedig egy kérés kiszolgálásakor.
- Attribútumai:
  - **page**: A beemelendő tartalom relatív URL-je.
  - **flush**: Ha true-ra állítjuk (a default false), a válasz puffert azonnal kiüríti.
- Példa:

```
<jsp:include page="/egyebek/fejlec.html"/>
```

Szabványos akciók: <jsp:forward>

- A kérést továbbítja statikus tartalomra, másik JSP oldalra, vagy servletre.
- Az aktuális JSP oldal végrehajtása megszakad, az oldal válasz puffere törlődik.
- Csak akkor használható, ha az oldal válasz pufferét még nem ürítettük, mert ellenkező esetben a már kiírt tartalmat nem lehet "visszaszívni", ekkor IllegalStateException dobódik.
- Attribútuma:
  - **page**: Az új tartalom relatív URL-je.
- Példa:

```
<% String inkabbIde = "/statikus/" + oldalNev + ".html"; %>
<jsp:forward page="<%= inkabbIde %>"/>
```

Szabványos akciók: <jsp:param>

- A <jsp:include> és <jsp:forward> akciókban használható paraméterek megadására a beemelendő, illetve átirányított tartalom lekérdezéséhez.
- Csak <jsp:include> és <jsp:forward> akciók törzsében fordulhat elő. (A tárgyalt akciók közül.)
- Attribútumai:
  - **name**: Az átadandó paraméter neve.
  - **value**: Az átadandó paraméter értéke.
- Példa:

```
<jsp:include page="/servletek/keres">
   <jsp:param name="q" value="ingyom-bingyom"/>
   <jsp:param name="maxresults" value="100"/>
</jsp:include>
```

Szabványos akciók: <jsp:element>

- Egy XML tag dinamikus létrehozására szolgál. XML vagy HTML tartalom létrehozásakor használható.
- Attribútuma:
  - ☞ **name**: A létrehozandó tag neve.
- A létrehozandó tag törzsét a <jsp:element> akció törzsében kell megadni.
- Példa:

```
<jsp:element name="<%= generaltNev %>">
   torzs
</jsp:element>
```

Szabványos akciók: <jsp:attribute> I.

- Két használata van:
  - Egy akció attribútumának megadása az akció törzsében.
  - Attribútumok megadása a <jsp:element> akció törzsében.
- Az attribútum értékét az akció törzsében kell megadni.
- Attribútumai:
  - **name**: Az attribútum neve.
  - **trim**: Ha true-ra állítjuk (ami a default), a törzs elején és végén lévő whitespace-t eldobja.

Szabványos akciók: <jsp:attribute> II.

- Példa az első használatra:

```
<entagjaim:valami>
   <jsp:attribute name="attrib">
      ertek
   </jsp:attribute>
</entagjaim:valami>
```

- A fenti megoldásnak például akkor van értelme, ha az akció dinamikus attribútumokat használ (lásd később).
- Példa a második használatra:

```
<jsp:element name="<%= generaltNev %>">
   <jsp:attribute name="attrib1">ertek1</jsp:attribute>
   <jsp:attribute name="attrib2">ertek2</jsp:attribute>
</jsp:element>
```

Szabványos akciók: <jsp:body>

- Akciók törzsének explicit megadására szolgál.
- Akkor van rá szükség, ha egy akció, vagy a <jsp:element> akcióval létrehozott tag legalább egy attribútumát <jsp:attribute> akcióval adtuk meg.
- Attribútumai nincsenek, a törzset az akció törzsében kell megadni.
- Példa:

```
<jsp:element name="<%= generaltNev %>">
   <jsp:attribute name="attrib">ertek</jsp:attribute>
   <jsp:body>az uj tag torzse</jsp:body>
</jsp:element>
```

Tag Extension I.

- A Tag Extension API lehetővé teszi saját akciók létrehozását.
- Az akciók tevékenységét Javaban kell leírni (bár a 2.0-ás JSP óta lehetőség van arra is, hogy JSP-ben írjuk le).
- Az akciók létrehozása három lépésből áll:
  - A **tag kódjának** elkészítése.
  - Egy **tag library leíró** készítése, ami tartalmazza az új tagek tulajdonságait.
  - A JSP oldalban egy **taglib direktíva** elhelyezése, amely az oldalon elérhetővé teszi a tag library-t.

Tag Extension II.

- A Tag Extension API fontosabb interfészei az alábbiak:
  - `Tag`, és a belőle származó
  - `BodyTag` és `IterationTag`.
- A fenti interfészek megvalósítását segítik az előre elkészített részleges megvalósítások:
  - `TagSupport` és `BodyTagSupport`.
- Az egyes interfészekből származó tagek célja eltérő:
  - `Tag`: egyszerű tagek,
  - `BodyTag`: olyan tagek, amelyek törzsüket manipulálni akarják,
  - `IterationTag`: olyan tagek, amelyek többször hajtják végre a törzsüket.

Példák tagekre: egyszerű tag I.

- Ez a tag egyszerűen egy szöveget ír ki.
- Először a tag kódját írjuk meg:

```
package tags;

import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class TestTag1 extends TagSupport {
   public int doStartTag () throws JspException {
      JspWriter out = pageContext.getOut ();
      try {
         out.println ("Ez a <strong>TestTag1</strong>.");
      } catch (IOException e) { throw new JspException (e); }
      return Tag.SKIP_BODY;
   }
}
```

Példák tagekre: egyszerű tag II.

- A következő lépés a tag library leíró elkészítése:

```
<taglib>
   <tlib-version>1.0</tlib-version>
   <jsp-version>2.0</jsp-version>
   <tag>
      <name>test-tag1</name>                          // a tag neve
      <tag-class>tags.TestTag1</tag-class>          // a tag osztálya
   </tag>
</taglib>
```

- Ezt a file-t a `WEB-INF` könyvtáron belül kell elhelyezni, a kiterjesztése rendszerint **tld**

Példák tagekre: egyszerű tag III.

- A deployment descriptorban megadhatunk egy virtuális URI-t, amely azonosítja a tag library leírót.
- Ezzel a módszerrel függetleníthetjük az alkalmazás többi részét a tld elhelyezkedésétől.

```
<web-app>
   <taglib>
      <taglib-uri>/testTaglib</taglib-uri>
      <taglib-location>/WEB-INF/test.tld</taglib-location>
   </taglib>
</web-app>
```

Példák tagekre: egyszerű tag IV.

- A tag kipróbálásához egy JSP oldalt is kell készítenünk:

```
<%@ taglib uri="/testTaglib" prefix="ttl" %>

<html>
   <head><title>Tag Test</title></head>
   <body>
      <ttl:test-tag1/>
   </body>
</html>
```

Példák tagekre: paraméterezhető tag I.

- Az alábbi tag attribútumként átvesz egy paramétert:

```
...

public class TestTag2 extends TagSupport {

   public int doStartTag () throws JspException {
      JspWriter out = pageContext.getOut ();
      try {
         out.println ("Ez a <strong>" + text + "</strong>.");
      } catch (IOException e) { throw new JspException (e); }
      return Tag.SKIP_BODY;
   }

   public void setText (String text) { this.text = text; }

   private String text;
}
```

Példák tagekre: paraméterezhető tag II.

- A text nevű attribútumot a tag library leíróban is definiálni kell:

```
<taglib>
   <tlib-version>1.0</tlib-version>
   <jsp-version>2.0</jsp-version>
   <tag>
      <name>test-tag2</name>
      <tag-class>tags.TestTag2</tag-class>
      <attribute>
         <name>text</name>
         <type>java.lang.String</type>
      </attribute>
   </tag>
</taglib>
```

Példák tagekre: paraméterezhető tag III.

- Használata:

```
<%@ taglib uri="/testTaglib" prefix="ttl" %>

<html>
   <head><title>Tag Test</title></head>
   <body>
      <ttl:test-tag2 text="paramtext"/>
   </body>
</html>
```

Példák tagekre: törzzsel rendelkező tag I.

- Az alábbi tag egyszerűen beemeli a törzsét:

```
...
public class TestTag3 extends TagSupport {
   public int doStartTag () throws JspException {
      JspWriter out = pageContext.getOut ();
      try {
         out.println ("Ez a <strong>");
      } catch (IOException e) { throw new JspException (e); }
      return Tag.EVAL_BODY_INCLUDE;
   }
   public int doEndTag () throws JspException {
      JspWriter out = pageContext.getOut ();
      try {
         out.println ("</strong>.");
      } catch (IOException e) { throw new JspException (e); }
      return Tag.EVAL_PAGE;
   }
}
```

Példák tagekre: törzzsel rendelkező tag II.

- Használata:

```
<%@ taglib uri="/testTaglib" prefix="ttl" %>

<html>
   <head>
      <title>Tag Test</title>
   </head>
   <body>
      <ttl:test-tag3>paramtext</ttl:test-tag3>
   </body>
</html>
```

Példák tagekre: a törzsét módosító tag

- Ez a tag csupa nagybetűsre konvertálja a törzsét:

```
...

public class TestTag4 extends BodyTagSupport {

   public int doAfterBody () throws JspException {
      JspWriter out = getPreviousOut ();
      try {
         out.println (getBodyContent ().getString ().toUpperCase ());
      } catch (IOException e) { throw new JspException (e); }
      return Tag.SKIP_BODY;
   }
}
```

Példák tagekre: iterátor tag

- Ennek a tagnek paraméterként megadhatjuk, hogy hányszor hajtsa végre a törzsét:

```
...
public class TestTag5 extends BodyTagSupport {

   public int doStartTag () {
      return iter == 0 ? Tag.SKIP_BODY : Tag.EVAL_BODY_INCLUDE;
   }

   public int doAfterBody () throws JspException {
      return --iter == 0 ?
            Tag.EVAL_PAGE : IterationTag.EVAL_BODY_AGAIN;
   }

   public void setIterations (int iter) { this.iter = iter; }

   private int iter;
}
```

Példák tagekre: dinamikus attribútumú tag I.

- Ennek a tagnek nincsenek előre meghatározott attribútumai:

```
public class TestTag6 extends TagSupport
      implements DynamicAttributes {

   public TestTag6 () { dA = new HashMap (); }
   public void setDynamicAttribute (String uri,String n,Object v) {
      dA.put (n,v);
   }
   public int doStartTag () throws JspException {
      JspWriter out = pageContext.getOut ();
      try {
         for (Iterator i = dA.keySet ().iterator ();i.hasNext ();) {
            String name = (String) i.next ();
            out.println (name + "=" + dA.get (name) + "<br/>");
         }
      } catch (IOException e) { throw new JspException (e); }
      return Tag.SKIP_BODY;
   }
   private HashMap dA;
}
```

Példák tagekre: dinamikus attribútumú tag II.

- Ezt a tényt a tag library leíróban is definiálni kell:

```
<taglib>
   <tlib-version>1.0</tlib-version>
   <jsp-version>2.0</jsp-version>
<tag>
      <name>test-tag6</name>
      <tag-class>tags.TestTag6</tag-class>
      <dynamic-attributes>true</dynamic-attributes>
   </tag>
</taglib>
```

Példák tagekre: változók visszaadása I.

- Egy tag létrehozhat script változókat is.
- Egészítsük ki az iterátor taget úgy, hogy a JSP oldalból le lehessen kérdezni a hátralévő iterációk számát. Ehhez létre kell hozni egy TagExtraInfo osztályt a taghez:

```
package tags;

import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class TestTag7TEI extends TagExtraInfo {
   public VariableInfo[] getVariableInfo (TagData tagData) {
      return new VariableInfo[] { new VariableInfo
            ("iter","java.lang.Integer",true,VariableInfo.NESTED) };
   }
}
```

Példák tagekre: változók visszaadása II.

- A tag a `PageContexten` keresztül férhet hozzá a változóhoz:

```
public class TestTag7 extends BodyTagSupport {

   public int doStartTag () {
      pageContext.setAttribute ("iter",new Integer (iter - 1));
      return iterations == 0 ? Tag.SKIP_BODY : Tag.EVAL_BODY_INCLUDE;
   }

   public int doAfterBody () throws JspException {
      pageContext.setAttribute ("iter",new Integer (--iter - 1));
      return iter == 0 ?
            Tag.EVAL_PAGE : IterationTag.EVAL_BODY_AGAIN;
   }

   public void setIterations (int iter) { this.iter = iter; }

   private int iter;
}
```

Példák tagekre: változók visszaadása III.

- A `TagExtraInfot` a tag library leíróban kell deklarálnunk:

```
<taglib>
   <tlib-version>1.0</tlib-version>
   <jsp-version>2.0</jsp-version>
<tag>
      <name>test-tag7</name>
      <tag-class>tags.TestTag7</tag-class>
      <tei-class>tags.TestTag7TEI</tei-class>
      <attribute>
         <name>iterations</name>
         <type>int</type>
      </attribute>
   </tag>
</taglib>
```

Példák tagekre: változók visszaadása IV.

- A tag használata:

```
<%@ taglib uri="/testTaglib" prefix="ttl" %>

<html>
   <head>
      <title>Tag Test</title>
   </head>
   <body>
      <ttl:test-tag7 iterations="5">
         Meg <%= iter %> iteracio van hatra.</br>
      </ttl:test-tag7>
   </body>
</html>
```