Web-based Distributed Systems

Idea

So far: Client is an application-specific program

- Client can be used by humans or
- Client can used to connect to other systems
- The functionality can easily be distributed in different ways between client and server

Disadvantage

- Updating the system results in modifications of the programs on the server as well as on the client

Now: We consider distributed systems with Web Clients

- Web Browser makes up the client
- Client is only used by humans
- User Interface is generated on the server
- The generated documents are displayed in the browser
- Advantage: Updates do not require modifications on the client-side („Zero-Deployment“)

Basic Standards

Communication-Protocols

- Web-Clients communicate using HTTP or HTTPS

Advantages

- widely used
- simple and robust
- not blocked by firewalls

Disadvantages:

- Connectionless (!)
- No transactions

Data format

- HTML or XML

Presentation-Format HTML

HTML is the classical language to describe Web-pages

For Web-based distributed applications it the most important technique to generate the user interface

Basic functionality of HTML: Formatting of static Web-pages

Today also used for dynamically generated pages and to allow the user to enter input

Structure of a HTML- Document

```
<HTML>
 <HEAD>
  <TITLE>Test-Document</TITLE>
 </HEAD>
 <BODY>
  <H1>Important header</H1>
  <P>Here we have text.</P>
  Now we have a <a href="xxx.html">Hyperlink</a>.
 </BODY>
</HTML>
```


Page-Description with HTML

HTML is simple ASCII-text. Formatting of text is performed by means of elements that are marked by tags

In its strict form an element is surrounded by start-tag <X> and an end-tag </X>

The interpretation of the tags and the presentation of the result on the screen is in the responsibility of the browser

As a result, the same HTML-page can be displayed differently by different browsers

HTML-Elements

Examples for HTML-Elements:

- <P> for paragraphs
- <H1>, ..., <H6> for headings
- <TABLE> for tables
- for emphasized parts

In addition to this we have elements addressing the actual formatting:

- <b> for boldface
- <FONT> for a particular font, color and size

User Input with HTML

Requirements on a Web-Client for distributed systems:

- Presentation of results
- Possibility to enter data by the user and send it to the server

Therefore we also need:

- HTML-Elements for input (Solution: Forms)
- A Mechanism to send the data from the client to the server (Solution: HTTP GET und POST)

URLs to Transport Data

URLs allow to transport data

Idea: encode the data as parameter within the URL and send this extended URL to the server

Format:

```
http://host:port/path/program?p1=v1&p2=v2
```

Parameter are supplied as name-value-pair

On the server-side the called `program` is processing the parameter

For this purpose it has to have an idea of the meaning of the parameter

HTTP GET and POST

A URL composed as described above is transported via HTTP GET to the server

Within a GET-message we combine the URL with all parameters

Alternatively we can use HTTP POST to transport the parameters separately from the URL within the body of a HTTP-Packet

Which operation is chosen can be observed by checking the URL in the browser:

- when using GET the URL contains all parameter
- when using POST we see only the part of the URL before the "?", all parameter are only within the message body

Advantage of GET: good for testing

Disadvantage: you might not want to let the user see the parameters; limited to 255 characters

Forms in HTML

HTML-Forms are enclosed in <FORM> and </FORM>

The parameter ACTION specifies which program has to get the parameters specified within the form

Within the form we can have different input fields (Text, Radio, Checkbutton, ...), that have a name

each form has a SUBMIT-button, to send the form

Example:

```
<FORM ACTION="http://www.uwa.edu.au/cgi-bin/whois.pl" METHOD="POST">
...
</FORM>
```


Example Form-Elements (1)

The Input-Element is often used without a closing Tag

Textfield:

- `<INPUT TYPE="TEXT" NAME="name_of_the_variable">`

Checkbox:

-
```
<INPUT TYPE="CHECKBOX" NAME="choiceA" VALUE="1">Choice 1
<INPUT TYPE="CHECKBOX" NAME="choiceA" VALUE="2" CHECKED>
Choice 2
<INPUT TYPE="CHECKBOX" NAME="choiceA" VALUE="3">Choice 3
<INPUT TYPE="CHECKBOX" NAME="choiceA" VALUE="4">Choice 4
```

Radio Buttom:

-
```
<INPUT TYPE="RADIO" NAME="choiceB" VALUE="1"> Choice 1
<INPUT TYPE="RADIO" NAME="choiceB" VALUE="2" CHECKED>
Choice 2
<INPUT TYPE="RADIO" NAME="choiceB" VALUE="3"> Choice 3
<INPUT TYPE="RADIO" NAME="choiceB" VALUE="4"> Choice 4
```

Submit Button:

- `<INPUT TYPE="SUBMIT">`

Example Form-Elements (2)

drop-down menu

```
<SELECT NAME="name_of_variable">
<OPTION> Option 1
<OPTION> Option 2
<OPTION> Option 3
</SELECT>
```

Example-Document

```
<HTML>
  <form action=http://localhost:8080/servlet/ProductSearch
    method="GET">
    <input type="text" name="product" value="Motor">
    <input type="submit" value="send request">
  </form>
</HTML>
```


Servlets

Servlet-Technology

Servlets can be viewed as a combination of two older approaches:

- CGI is a technique used at the server-side to supply parameters to a program
- Applets are Java-programs, that reside on the Web-server but are loaded to the client to be executed

Servlets are Java-programs, that are executed within the server

Servlet-Engine

To use servlets we need a servlet-engine (also called servlet container) that is able to invoke the servlets and hand-over the parameters provided by the client

There exist a lot of different servlet-engines

We will use Jakarta Tomcat (http://jakarta.apache.org)

Tomcat can be

- used standalone as a combination of Web-Server und servlet/JSP-container or
- used in combination with other web-servers like Apache or Microsoft IIS

Properties of the Servlet-Technology

Servlets can be used by means of Java class-libraries

The API provides objects for

- describing a servlet
- representing a client-request and
- encoding the reply to the client

In addition to this we can use mechanisms to maintain a state to overcome the lack of this concept in HTTP

Classes/Interfaces

The servlet-library is in the following packages:

- javax.servlet.* und
- javax.servlet.http.*

Most important interface: Servlet

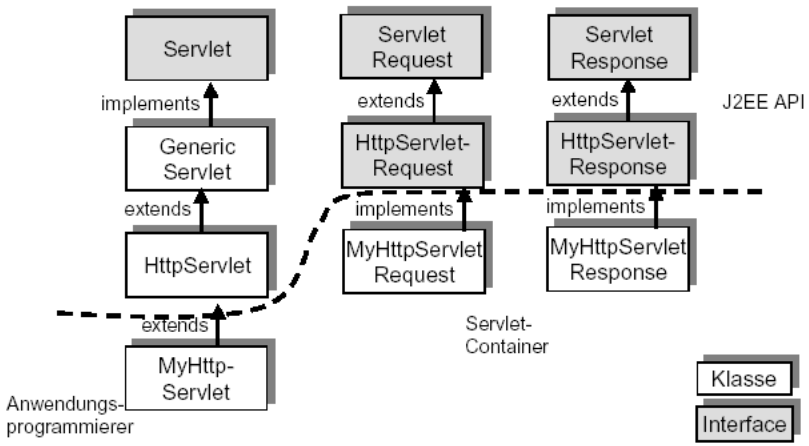
Generic classes / interfaces

- GenericServlet
- ServletRequest
- ServletResponse

Specific web-servlets-classes / interfaces

- HttpServlet
- HttpServletRequest
- HttpServletResponse

Servlet: Classes/Interfaces



Creating own Servlets

Servlets can be created using one of three possible approaches:

- We implement the interface Servlet („implements“).
- We create a subclass of GenericServlet („extends“).
- We create a subclass of HttpServlet („extends“).

The choice of the variant depends on the specific task

In most cases the preferable way is to extend the class HttpServlet

The Interface Servlet

Each servlet has to implement this interface

- this can also be done indirectly by using the base-class GenericServlet or HttpServlet

Why?

- The servlet-engine depends on the existence of the methods in this interface
- The interface servlet can therefore be considered as a part of a requirements specification for a servlet

Important Methods of Servlet

void service(ServletRequest req, ServletResponse res)

- to handle a request
- is called by the servlet-container

void init (ServletConfig config)

- to start a servlet
- is called by the servlet-container

void destroy()

- to destroy a servlet
- is called by the servlet-container

Life Cycle of a Servlet

Servlets are executed within a servlet-engine that invoking methods of the servlet

When the servlet is created the method init() is called

Afterwards for each request of the client the method service() is called

- is invoked each time on the same instance!
- a separate thread for each method invocation

Finally the method destroy() is called

Generic Servlets

GenericServlet is the simplest implementation of the interface Servlet

init() and destroy() are implemented, but there is no implementation for service()

- as a result the class is abstract
- we can not instantiate objects but have to create a subclass

The class be used for all kinds of applications; however, for applications that make use of the specific properties of HTML and HTTP, it is recommended to use HttpServlet

Structure of a Generic Servlet

```
class NeuesServlet extends GenericServlet {
    // service() has to be implemented
    public void service( ServletRequest req,
     ServletResponse res) {
     // read req
     // perform task
     // write res
    }
}
```


HttpServlet

The class HttpServlet is optimized to be used in the context of HTTP/ HTML

In contrast to GenericServlet the method service() is already implemented

service() is calling the specialized methods doGet() and doPost(), that correspond to the HTTP-operations GET and POST

As a result, instead of service() we have to implement doGet() and doPost()

Handling a Request submitted with GET

```
class NeuesServlet extends HttpServlet {
    //...
     public void doGet( HttpServletRequest
          req, HttpServletResponse res) {
     //...
     res.setContentType("text/html");
     PrintWriter sout = res.getWriter();
     sout.println("<H1> output of a
     Servlet</H1>");
     //...
     }
}
```


The Generic Request-Object

By means of using the interface ServletRequest the servlet gets access to the parameters supplied by the client

The most important methods of this interface:

- `java.util.Enumeration getParameterNames ()`
  Returns an Enumeration that contains the strings with the names of the parameters supplied by the client
- `String [] getParameterValues(String name)`
  Returns an array of strings that contains the values of the parameter. If the parameter does not exist, null is returned
- `String getParameter (String name)`
  Returns the value of the parameter. If the parameter does not exist, null is returned

Example: Reading all Parameters

```
public void service( ServletRequest req,
            ServletResponse res) {
   // ...
   Enumeration e = req.getParameterNames();
   while (e.hasMoreElements()) {
      String name = (String) e.nextElement();
      String werte[] = req.getParameterValues(name);
      System.out.print(" Name:"+ name+" Values:" );
      if (werte != null)
         for (int i= 0; i< werte.length; i++)
            System.out.print( werte[i] + " ");
      else
         System.out.print("( none)");
      System.out.println();
   }
} // End of service()
```


Reading for a simple Case

When we know the name of the parameter and the parameter has just single value, a simpler approach is possible:

```
public void service( ServletRequest req,
      ServletResponse res) {
   // ...
   String vorname = req.getParameter("vorname");
   String nachname = req.getParameter("nachname");
   System.out.println("Guten Tag, "+ vorname +
                                          nachname);
}
```


Additional Methods of ServletRequest

ServletRequest enables access to addressing information:

- `String getServerName()` returns the name of the machine on which the own server is running
- `int getServerPort()` returns the corresponding port of the own server
- `String getRemoteAddress()` returns the IP-address of the client
- `String getRemoteHost()` returns the name of the machine of the client

HttpServletRequest

This interface is used for similar purposes as the generic ServletRequest

There exist extensions for the HTTP- / Web- environment:

- Access to HTTP- variables (URL, ...)
- Management of Cookies and Sessions

The Interface ServletResponse

This is the counterpart to the request-object: it is used to send the response of the server back to the client

Approach: ServletResponse allows access to streams into which the servelt can write

In addition to this some administrative functionality, e.g. the specification of the MIME-type

The most important Methods

`void setContentType(String type)`

Sets the Mime-type of the response

`java.io.PrintWriter getWriter ()`

Returns a PrintWriter-object that is used to send text to the client

`ServletOutputStream getOutputStream ()`

Returns a ServletOutputStream-object that is used to send binary data to the client

Example: Text to the Client

```
public void service(ServletRequest req,
     ServletResponse res) {
   //...
   res.setContentType("text/plain");
   PrintWriter pw = res.getWriter();
   pw.println("Sending von ASCII- Text");
   pw.flush();
}
```


HttpServletResponse

Again many similarities to the generic interface (in this case ServletResponse)

In addition, we have for example the possibility

- to send status information to the client (sendError()),
- to set the header of a HTML-document (setHeader()),
- to set cookies on the client-machine (addCookie()).

Cookies

Cookies can be used to store information about a client on the client-machine

Purpose: Reuse of this information when the client sends additional requests

Example: Amazon welcomes you with your name

Servlets set as well as read cookies

For this purpose we have the methods

- addCookie() in HttpServletResponse
- getCookies() in HttpServletRequest

Example: Set a Counter

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws IOException, ServletException
    {
        PrintWriter out = response.getWriter();
        Cookie c = new Cookie("TestCookie","0");
        // add cookie to the response-object
        response.addCookie(c);
        // and write to the stream
        out.println("Cookie gesetzt");
    }
```


Example: Read the Counter

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws IOException, ServletException
    {
       PrintWriter out = response.getWriter ();
       Cookie[] cookies = request.getCookies();
       if (cookies == null){
         out.println("No Cookies available");return;
       }
       Cookie c = null;
       for (int i=0; i<cookies.length; i++) {
        if(cookies[i].getName().equals("TestCookie")) {
          c = cookies[i]; break;
        }
       }
       if (c != null) {
         String value = c.getValue();
         out.println("Cookie found: " + value);
       }
    }
```


Disadvantage of Cookies

Cookies have a bad reputation as they allow to store arbitrary information on the hard disk of the user

The user can therefore disable the possibility to store cookies

Another solution to store information about the client (but only for a short period of time) is to use sessions

- Sessions can be realized based on cookies but it also possible to use sessions without cookies

Sessions

The Servlet-library contains a class called HttpSession

Sessions are used to store information about clients but this time the information is stored on the server

Like for a cookie, we can store in a session parameter and their corresponding values. However, in a session the values can be arbitrary objects and not just strings as in cookie

When a client-request is received, it is checked whether for this client a session is already existing

- if yes, the existing session is used
- otherwise a new one is created

Important Methods of HttpSession

- `java.util.Enumeration getAttributeNames ()`
  Returns an Enumeration of Strings for the names of the attributes in the session
- `Object getAttribute (String name)`
  Returns the value of the attribute. Returns null, if the attribute is not existing
- `void setAttribute (String name, Object value)`
  Creates a new attribute of the session. An object is bound to the name of the attribute
- `void removeAttribute (String name)`
  Removes the attribute with the name provided as a parameter
- `void setMaxInactiveInterval(int interval)`
  Specifies the number of seconds for the timeout interval. If the client is inactive for this interval the session is terminated
- `void invalidate()`
  Terminates the session.

Example: Storing Client-Data

```
public void doGet(HttpServletRequest req,
                  HttpServletResponse res)
    throws IOException, ServletException
  {
      HttpSession session = req.getSession();
      Enumeration er = req.getParameterNames();
      while (er.hasMoreElements()) {
            String name = (String)er.nextElement();
            String value = req.getParameter(name);
            session.setAttribute(name, value);
      }
  }
```


The Session-Number

How can the client recognize that a session for this client is already existing?

- A Session-ID is used to distinguish sessions
- This Session-ID is generated by the server and send to the client
- The client has to return the Session-ID in each request to the server

How does the client manage the Session-ID?

- At first, we try to store the Session-ID in a cookie
- If the client does not allow cookies, the Session-ID is encoded in the URL (URL-Rewriting)

When a new session is created, we use at first both techniques

- if the client accepts cookies, no URL-Rewriting is necessary for further requests
- Otherwise it is continued

URL-Rewriting

By means of URL-rewriting the Session-ID is send in the next request to the server. The URL

- http://localhost:8080/a6/SessionExample

is for example transformed into

- http://localhost:8080/a6/SessionExample;jsessionid=4DE575CFD8EB45D04A5E341ADD329EA4

The actual format of the rewritten URL depends on the used servlet-engine

To enable URL-rewriting, we have to make an explicit request in our code

Example: URL-Rewriting

```
public class SessionBeispiel extends HttpServlet {
  public void doGet(HttpServletRequest req,
      HttpServletResponse res)
      throws IOException, ServletException
  {
   HttpSession session = req.getSession();
   res.setContentType("text/html");
   PrintWriter out = res.getWriter();
   out.println("<html><body>");
   out.println("<a href=\"" + res.encodeURL("bla") +
      "\">hier klicken</a>");
   out.println("</body></html>");
   // ...
  } // doGet
} // SessionBeispiel
```