Programming for the Java Platform Enterprise Edition

Javs Server Pages (JSP) I

Opracował:
mgr inż. Tomasz Jastrząb

Table of contents

1 Introduction

Java Server Pages (JSP) is a technology related to the data presentation layer, working on the server side. Unless it uses some Java EE platform-specific elements it does not require a Java EE compliant server. JSP, being a part of data presentation layer, requires also a layer responsible for data handling, which in case of web applications is usually based on Java Servlets technology.

Aforementioned layers constitute parts of MVC (Model-View-Controller) design pattern implementation, which defines the way of dividing application functionality into business logic layer (Model), data presentation layer (View) and an intermediate layer, responsible for data transfer between the other two layers (Controller). In case of web applications the Model part is represented by JavaBean classes, created in servlets based on HTTP parameters obtained from HTML/JSP pages. Data presentation layer, as mentioned before, is composed of JSP pages, adjusting their contents to the data sent from servlet. Servlet itself becomes the Controller part.

1.1 Java Servlets basics

Servlets, similarly to JSP pages, work on the server side, being responsible for processing requests and generating responses. Since they are usually used together with HTTP protocol, they frequently extend *javax.servlet.http.HttpServlet* class, although it is only necessary to implement *javax.servlet.Servlet* interface.

Servlet's life cycle is described by three methods: *init*, *service* and *destroy*. The *init* method is used for servlet initialization and usually can be used for the initialization of resources that shall be available throughout the lifetime of a servlet inside servlet container. An example of such a resource can be a database connection, in the case of low-level database communication mechanisms use (i.e. JDBC, Java DataBase Connectivity). Analogously the *destroy* method is called at the time the servlet is released from the container and may be used for resources cleanup (e.g. connection closing).

The solution keeping long-lasting open database connection has its drawbacks, especially in case of multi-threaded applications, but these will not be mentioned here. Furthermore, instead of overriding the *init* method, one may use so called listeners, whose methods are called in characteristic moments of servlet's life cycle (e.g. when servlet context is created – at startup stage, when attributes are added or removed, etc.).

The main part of servlet's life cycle is the time of requests handling, i.e. the *service* method. However, when using HTTP protocol one usually does not override the *service* method directly, but instead overrides methods responsible for particular request types (e.g. GET or POST, which are handled by *doGet* and *doPost* methods). The request and response related to a servlet are represented by implementations of *javax.servlet.ServletRequest* and *javax.servlet.ServletResponse* interfaces, or in case of HTTP protocol *HttpServletRequest* and *HttpServletResponse* from javax.servlet.http package. Objects implementing these interfaces are passed as arguments among others to *doGet* and *doPost* methods (and also to the other methods handling various HTTP request types).

As a remainder, GET method causes the request parameters to be concatenated to the URL address, after '?', connected by '&' (in case there is more than one parameter). Parameters passed this way constitute key-value pairs, where the key is the parameter name, e.g. from an HTML form. Requests of this type have limited length and due to obvious reasons cannot be used for secure data transfer. The GET method is the default method for HTML forms. The POST method in turn uses HTTP headers for data transfer, which means it is free from data size limitations and the dangers related to the GET method.

1.1.1 HttpServletRequest

HttpServletRequest represents a request sent to server, e.g. from an HTML form, JSP page, other servlet or directly from the web browser. Among the most important methods available through *HttpServletRequest* the following may be listed:

- *getParameterNames*, *getParameter* – these methods allow for obtaining the parameters sent with the request. The first method returns a collection of parameter names, which may be used for verification which parameters were sent. The other method allows for reading the value associated with given parameter name, returning null in case no parameter with given name was sent.
- *getHeaderNames*, *getHeader* – these methods allow for obtaining the headers sent with the request and they work analogously to their parameter-related counterparts.
- *getAttribute*, *setArribute* – these methods allow for getting and setting the attributes of the request, used usually together with the methods forwarding current request to some other servlet or JSP page.
- *getSession*, *getCookies* – these methods can be used for accessing the mechanisms storing client data inside session or cookies.

The request object is most frequently used for getting request parameters, and based on their values performing further processing. Attribute-related methods are particularly useful in case of MVC (Model-View-Controller) design pattern implementation, where the Model objects can be inserted as attributes and forwarded to the View part.

1.1.2 HttpServletReponse

HttpServletResponse represents the server response, which depending on the response type set can be e.g. a plain text, an HTML text or an image. Among the most important methods the following can be found:

- *getHeaderNames*, *getHeader* – similarly to the case of *HttpServletRequest* these methos allow for access to the headers, but this time they are connected with the response, not the request.
- *setContentType* – this method allows for setting the MIME type of the response. Types are passed as *String* objects„ e.g. "text/html", which denotes an HTML text page.
- *getWriter*, *getOutputStream* – depending on the type of response set (textual or binary one) these methods allow for retrieving the correct output stream filled with the response. Most often this will be the character (textual) stream.
- *sendError*, *sendRedirect* – these methods allow for redirections, to standard error pages (e.g. with status code 404) or some other pages (based on URL address), respectively.
- *addCookie* – this method allows for adding new cookie to the response. If cookies are enabled in the browser they should be resent with successive requests.

1.1.3 Redirections

Servlets allow for generation of four types of redirections, differing in the way they work and the situations they can be applied to. Respective redirection types will be described in successive paragraphs.

sendError As mentioned earlier, while discussing *HttpServletResponse* interface (section 1.1.2), *sendError* method allows for redirecting to standard error page. The errors handled this way are related to the predefined response status codes, e.g. code 404 meaning "resource not found", code 400 denoting bad request, and others. Respective codes are defined as constants in *HttpServletResponse* interface, and may be passed to the *sendError* method. This method takes as an optional argument a message to be displayed on the error page giving further details on the error. After the redirection, no further writing to the response should be performed.

sendRedirect *sendRedirect* method was also mentioned in section 1.1.2. It allows for redirection to some web resource, whose URL address is given as an argument to this method. The characteristic feature is that the location given as an argument is independent of current location, i.e. no request or response is passed with the redirection. Similarly to *sendError*, *sendRedirect* enforces that no further writing to the response is done.

forward *forward* method is defined in *RequestDispatcher* interface, which in turn can be obtained using the servlet context (*getServletContext* method and later *getRequestDispatcher*). It allows for redirection to some other resource, given as an argument to the *getRequestDispatcher* method, but unlike *sendRedirect*, current request and response are passed along with the redirection.

The aim of this method is to pass the request object further, after modifying it by attribute insertion (through *setAttribute* method), but before the response was committed. The target of redirction can be another servlet or a JSP page, residing on the same server.

include A method that is somehow symmetric to the *forward* method is the *include* method, also being defined in *RequestDispatcher* interface. It is used for passing the response object further, or to be more precise, for including some other resource in current response. It may constitute a way of repeatable, parametrized inclusion of the parts of code placed in some other servlet or JSP page.

1.2 JSP basics

Java Server Pages constitute a way of combining HTML and Java, although the use of Java should be restricted to the data presentation functionality, not business logic implementation. The basic difference between HTML and JSP is that the latter allows for dynamic modification of page contents, while the former is usually used for static content generation.

JSP pages are translated to servlets, thus some predefined objects are accessible in every JSP page. Also, all HTML parts are automatically packed into methods responsible for servlet response generation, being usually *print* or *println* methods. Among the predefined objects the following may be distinguished, among others:

- *out* – it is the output stream filled with data to be displayed as a response. It is obviously not the standard output stream, but the stream used for server response generation,
- *request*, *response* – request and response objects, available as arguments of *doGet* and *doPost* servlet methods,
- *session* – session object, of *HttpSession* type,
- *exception* – exception object, available only in so called diagnostic pages, i.e. ones marked as "error page".

JSP syntax is composed of three groups of elements, described in successive sections, namely: directives, tags and scriplets.

1.2.1 Directives

Directives are usually placed at the beginning of a JSP document and are enclosed within `<%@` and `%>` markers. JSP defines three types of directives, namely *page*, *include* and *taglib*. Among these directives, *page* directive appears in almost every JSP page. Every directive possesses also some attributes, which may have default values.

The *page* directive may, among others denote the MIME type of page contents and its encoding (in *contentType* attribute). Moreover it may be used for package imports (with *import* attribute), exactly in the way packages can be imported using `import` keyword in case of plain Java classes. Furthermore this directive can be used for denoting the diagnostic page address (*errorPage* attribute) or for marking given page as an error page (*isErrorPage* attribute). Error page is a page which is used automatically in case some uncaught exception occurs.

The *include* directive can be used for external HTML or JSP files inclusion, but the file is included at the translation stage. It means that any changes introduced in the external file, to be visible in the output page require repeated translation. This directive contains single attribute called *file* specifying the resource to be included.

The *taglib* directive can be used for adding tag libraries, which may be later used in the JSP page. To do so, it is necessary to define both attributes of this directive, namely *uri*, denoting the path or address of the tag library, and *prefix*, specifying the prefix string appearing before tags. The details of tag libraries will be discussed in the next instruction.

1.2.2 Tags

Standard JSP tags are placed between < and >, similarly to HTML tags, and they begin with jsp prefix. Altogether there are 7 tags, among which the following can be found:

- <jsp:forward> – a counterpart for the *forward* method from *RequestDispatcher* interface, it allows for redirection to another resource being for instance a JSP page or a servlet. The location to which the redirection is performed is given by *page* attribute. It is also possible to specify some additional parameters putting them inside <jsp:param> tags. They can also overwrite the parameters sent inside request object, if they have the same names[1].

- <jsp:include> – a counterpart for the *include* method from *RequestDispatcher* interface, it allows for some resource inclusion in the generated response. Similarly to the <jsp:forward> tag it may specify additional parameters inside <jsp:param> tag. Unlike the *include* directive, resource mentioned in this tag is included with every request and thus changes in the resource do not require any additional effort or translation to be made visible.

- <jsp:useBean> – this tag is used for creation or binding of JavaBean[2] classes inside JSP page. The use of <jsp:useBean> tag is connected with giving an identifier to the bean instance (*id* attribute) and specifying its class or type (*class* and *type* attributes respectively). In case of class specification (including the package) a new object is created, using the parameterless constructor (unless one exists), and in case of type specification an existing object is used, or exception is thrown if one is not found. It is also possible to specify the scope of bean life time, using the *scope* attribute, which may take values: **page**, **request**, **session** or **application**.

- <jsp:getProperty>, <jsp:setProperty> – these tags are used together with the <jsp:useBean> tag and allow for reading and writig the properties of given JavaBean class. Both of them contain the *name* attribute, whose value corresponds to the value given in the *id* attribute of the <jsp:useBean> tag. Both of them contain also the *property* attribute

[1]To be more precise, parameters sent using <jsp:param> tag are put in front of the parameters table, thus the impression of overwriting parameter value in case of *getParameter(...)* method use. In such a case *getParameterValues(...)* method is recommended.

[2]JavaBean classes are ordinary Java classes with parameterless constructor as well as getters and setters for respective private fields (properties).

specifying the name of given property. Moreover the <jsp:setProperty> tag contains also *param* and *value* attributes used for specification of a request parameter name bound to given property, or a direct value assigned to given property. In case of the <jsp:setProperty> tag, *property* attribute may take the wildcard value of '*', which means that request parameters and JavaBean properties will be bound automatically based on the same names.

1.2.3 Scriplets

Scriplets constitute parts of Java code introduced into the generated servlet. depending on the markers enclosing given part of code it may be treated as a comment, a declaration, an expression or simply as a scriplet. Comments in JSP pages are enclosed in <%--, --%> and are treated similarly to the Java comments, i.e. they do not constitute part of actual code.

Declarations, placed between <%!, %>, may involve both variables and methods, although in case of methods it is better to place them in a separate Java class. Variables are created as servlet fields, not as local variables, so their visibility level may be marked as well. Declarations are considered at the translation stage.

Expressions are put between <%=, %> and are automatically converted to textual form (i.e. *String* objects) and then placed inside *println* method calls from the level of predefined *out* object.

Scriplets are direct Java code fragments inserted into the page, which may be interleaved with HTML tags. They are placed inside <% and %> markers. Although they are available, their use should be avoided, so that basic role of JSP pages is not lost, i.e. the role of data presentation layer. Particularly, they should not be used for the purpose of business logic implementation.

2 Laboratory

The description enclosed in this section is based on Netbeans 7.1.

2.1 Project configuration

To create a web application project using both servlets and JSP technology it is necessary to select menu ***File* → *New Project...***, and then *Java Web* → *Web Application*. After naming the project it is necessary to select a server and Java EE version, although for the purpose of servlets and JSP use one should **not** add the project to any enterprise application. It is also not necessary to select any frameworks, so the choices can be confirmed using **Finish** button. By default, the project that gets created, apart from *Source Packages* folder, should contain *Web Pages* folder, and inside of it *WEB-INF* folder and welcome JSP file called *index.jsp*.

Web Pages folder is dedicated to storing JSP pages as well as other web-related contents, except for servlets (e.g. HTML pages, images, etc.). *WEB-INF* folder in turn is related to storing data that should not be directly visible from outside, e.g. servlet configuration data in the form of *web.xml* file (since Servlet 3.0, this file is not necessary).

To gain the access to a servlet it is necessary to define the mapping properly. In case of newer version of servlets specification *@WebServlet* annotation may be used with its *urlPatterns* attribute specified. In older versions it was necessary to define <servlet-name>, <servlet-class> and <url-pattern> tags in *web.xml* file.

Servlet creation can be performed by clicking right mouse button (rmb) on the level of *Source Packages* folder, and then selecting ***New* → *Servlet***. In the next steps it is necessary to specify servlet name in *Class Name* field and its package (*Package*). After pressing **Next** a URL mapping should be given (*URL Pattern(s)*), optional initialization parameters (*Initialization Parameters*) and information about configuration inclusion in *web.xml* file (option *Add information to deployment descriptor (web.xml)*). Finally **Finish** button should be clicked to confirm inserted data.

HTML or JSP pages creation can be done using rmb on the level of *Web Pages* folder, and then selecting ***New* → *HTML*** or ***New* → *JSP***. In both cases the name of the file should be given (in the field marked *HTML File Name* or *File Name*) optionally with the name of a subfolder that will be created/selected inside *Web Pages* folder. In case of JSP pages standard syntax should be selected (*JSP File (Standard syntax)*).

Running both servlets and JSP files requires server (e.g. GlassFish) startup for instance using the *Services* → *Servers* tab, and later in the web browser typing in the address `http://localhost:8080/`, followed by application name (usually) and servlet or HTML/JSP page address, depending on the mapping (for servlets) or location in folder structure (for HTML/JSP pages). To run the elements properly it is necessary for the servlets, HTML or JSP pages to be deployed on server, which may be achieved by choosing ***Deploy*** option on the project level. Alternatively given file may be run using rmb and then ***Run File***.

3 Tasks

The tasks mentioned in this section are supposed to be performed during the laboratory classes, except for task no. 4 which defines the requirements for the report.

3.1 Task no. 1

Create a JSP page displaying current time in the format "HH/mm/ss", where H stands for hour, m minute, and s second.

Create a servlet displaying text "Hello, world. Current time is" and the current time. To obtain current time use previously created JSP page and *include* method.

Create a JSP page displaying text "Hello, world. Current time is" and the current time. To obtain current time use previously created JSP page and *include* directive as well as <jsp:include> tag. Compare the behaviour of the JSP page and the servlet.

Modify the format of displayed time so that it uses '-' instead of '/'. Rerun again the servlet and JSP page and observe the differences in their behaviour related to external JSP page inclusion.

Create JSP page, which gets the value of an attribute named *username* and displays it in the browser.

Create a servlet which gets a request parameter named *name*, containing user name (the parameter may be passed using the GET method by concatenating it to the servlet's URL). Servlet should verify:

- whether the parameter was passed,
- whether the parameter is not empty,
- whether the name starts with uppercase letter.

In case of correct data the servlet should set the request attribute named *username*, and then generate redirection to previously created JSP page, using *forward* method. In case of incorrect data, attribute *username* should contain an error message.

Create a JSP page performing the functionality of the servlet mentioned above and using <jsp:forward> tag for performing redirection. Compare the behaviour of both redirection types.

3.2 Task no. 2

Create a JavaBean class named Person, with fields *firstName*, *lastName*, and *age*.

Create a servlet which obtains data related to Person object from an HTML form (using POST method). The servlet should create three Person objects – the first one should make the first and last name lowercase and insert the object as request attribute; the second one should increment the age by a random value in the range [1, 10] and insert the object as session attribute; the third one should make the first and last name uppercase and insert the object as servlet context attribute (*ServletContext*). Generate the redirection to JSP page using *forward* method.

Create JSP page using <jsp:useBean> tag instantiating new Person object and filling it with request parameters' data (use <jsp:setProperty> tag). Define the scope of the object as **page**. Use <jsp:useBean> tag to bind existing objects of a Person class with the scopes **request**, **session** and **application**. Display the properties of all JavaBeans using <jsp:getProperty> tag. Verify which objects were bound to respective scopes and find which scope is the default one.

3.3 Task no. 3

Create an application managing an internet shop using MVC design pattern. To do so create JavaBean class representing single Product (e.g. having fields *name*, *price*, *quantity*). Create HTML forms allowing for specification of data for:

- new product creation,
- product removal based on its name,
- product properties update based on its name.

Create a JSP page displaying current product list and a servlet that gets the data from respective forms and manages the list of products. The servlet should play the role of a controller, which depending on the information provided from the forms should perform appropriate operations on the list of products (stored in Warehouse object constituting part of application Model). The servlet should also redirect the user to the page displaying current product list, or to the error pages in case of incorrect data given. JSP page should use expressions (i.e. <%=, %> markers).

3.4 Task no. 4

Create business application using web client view composed of JSP pages and a servlet playing the role of a controller. The application should communicate with the session beans created in the previous report.

- data constituting the contents of entity classes should be provided by the user through HTML forms,
- forms should use POST method,
- erroneous data should be detected on the servlet side and corrected before performing database operations,
- in case of incorrect data user should get redirected to an error page, which should contain a back-reference (link) to the initial page,
- session beans may be accessed through dependency injection or JNDI lookup in the servlet,
- report should contain Javadoc documentation for all created elements (classes, fields, methods),

The report should consist of three folders: *bin*, *doc*, *src*. *bin* folder should contain an *.ear file containing the enterprise application. *doc* folder should contain generated Javadoc documentation including *@author* and *@version* tags on the class level as well as private and package private members. *src* folder should contain source codes with packages. Inside *doc* and *src* folders the elements related to every part of the application should be placed (in separate subdirectories), including bean-related part, application client and the console part as well.

Identifiers of created classes, fields and methods should be written in English and comply with Java naming convention.