Java Servlets

Basic Concepts and Programming

Giuseppe Della Penna

Università degli Studi di L'Aquila
giuseppe.dellapenna@univaq.it
http://people.disim.univaq.it/dellapenna

> *This document is based on the slides of the Web Engineering course, translated into English and reorganized for a better reading experience. It is not a complete textbook or technical manual, and should be used in conjunction with all other teaching materials in the course. Please report any errors or omissions to the author.*

1. Java Servlet Basics

1.1. Introduction to Servlets

Servlets are special Java classes that run in specific web servers, called **servlet containers**.

Servlets are exposed as a standard web resources (i.e., they can be referred to using a URL).

Servlets act in the traditional *request/response* way, typical of the *server-side scripting*: when the user requests a servlet via the associated URL, the server activates it, executes it, and returns the result as the content of the resource.

Java **Servlet APIs** allow one to program the servlet behavior without any knowledge of the server/client characteristics and transfer protocol.

The code is standard Java code, which can make use of all the libraries and utilities for language, including the connection to all DBMS via JDBC, the use of XML through JAXP, etc..

Servlets replace CGI providing a high degree of safety, versatility and abstraction to the programmer.

1.2. Where and How to Run a Servlet

To run a servlet, a particular server is required which can serve as **servlet container**, providing adequate support to their activation and execution.

The most used *free* servlet container is **Tomcat** (Apache).

The traditional Apache HTTPD server is not suitable for containing servlets, but it is possible to install it side-by-side to a Tomcat installation via a suitable *connector*.

Tomcat is a *lightweight* container, and **implements only the basic technologies** for the development of web applications (*Servlets*, *JSP*). The complete **JEE Web profile** is available in more complex servers as *TomEE*, which is based on Tomcat, or *Glassfish*.

Warning: since version 10, Tomcat has been updated to support the evolution of Java EE, i.e. Jakarta EE: previously written web applications cannot run on Tomcat 10 unless they are adapted, mainly by modifying the packages of javax.* classes to jakarta.* (see https://tomcat.apache.org/migration-10.html)

Apache Tomcat Configuration

Apache Tomcat is available for all platforms (it is itself a Java program) and can be downloaded from http://tomcat.apache.org/.

The installation of Tomcat can be assisted by an installation script, but normally the application is distributed as a compressed archive that you just need to unzip and use by calling the appropriate scripts in the *bin* directory.

You can choose to automatically start the server as a **service** (Windows) or **daemon** (UNIX), or **manually**. Of course, *on a development machine, manual start-up is preferred*.

If the server is launched directly, and not through an IDE, the correct version of the JDK must be discoverable by the Tomcat startup script. To this aim, make sure that the **JAVA_HOME** environment variable is correctly set.

Tomcat uses two main folders for its execution: **CATALINA_HOME** contains the server binaries and libraries, while **CATALINA_BASE** is the area with user data, including installed web applications and server logs. The default for **CATALINA_BASE** is the same value as **CATALINA_HOME**. However, it is useful, and in some cases necessary, to define a **CATALINA_BASE** that points to a folder in your *home directory* (while **CATALINA_HOME** will probably be inside the applications folder of your operating system). In this way, you have an ad-hoc environment in which to develop and test applications, totally under your control.

Once executed, the default Tomcat instance responds on port **8080**.

Via the url http://localhost:8080/manager/ you can configure the server through a web application. and monitor the status of the web applications running in the server.

To access such administrative applications, you should first **create a user with administrative privileges**, adding to the **CATALINA_BASE/conf/tomcat-users.xml** file a line like the following

```xml
<user username="admin" password="adminpass" roles="admin,manager"/>
```

Many IDEs, such as Netbeans, automatically create an ad-hoc **CATALINA_BASE** and configure a fictitious user within it that will be used to allow the IDE itself to control Tomcat, making the operations of starting the server and deploying applications completely transparent to the user. In a development setup, especially when you are learning how to develop on such a platform, this is the most practical and suitable solution.

1.3. Web Application Context

Web applications are executed in **contexts**. In general, each context corresponds to a particular directory configured on the server and associated with a specific URL.

Each context has a set of associated system-defined and user-defined *attributes* (we will see how such attributes can be used, e.g., to configure the web application) accessible through the **ServletContext** object, and can be configured to *run some code when the application is started* (i.e., the web application is deployed or the server where it has been deployed is started) *or stopped* (i.e., the web application is undeployed or the server where it has been deployed is stopped) using **ServletcontextListener** objects.

To manually create a new web application it is sufficient to create a subdirectory in the **CATALINA_BASE/webapps** directory of Tomcat. The context name will be the one of the directory.

At this point, to test the new context, you can insert a plain html file in the directory and try loading the URL http://localhost:8080/CONTEXT_NAME/FILENAME, where *CONTEXT_NAME* is the name of the created subdirectory. For example http://localhost/project/index.html

However, in order to make a fully functional web application, we also need to prepare a special subdirectory structure in the context, and write some configuration files. The main elements of this structure are discussed below.

However, **the recommended installation method for web applications is to use an IDE (e.g., Netbeans) to create the application and package it in a *war* file (Web ARchive), which can then be *manually* copied into the *webapps* Tomcat directory or *automatically* deployed by the IDE**.

1.4. Web Application Directory Structure

Directories associated to a web application have a particular base structure which enables the server to access dynamic (servlets, JSP) and static (html, css, images, etc..) resources. In particular:

- The **WEB-INF** subdirectory contains some configuration files, including the *web application deployment descriptor* (web.xml).
- The **WEB-INF/classes** subdirectory contains the Java classes of the application, including servlets. Following the Java conventions, individual classes should be placed in a directory structure corresponding to their package name.
- The **WEB-INF/lib** subdirectory contains the JAR libraries used by the application, including the third party ones, such as JDBC drivers.
- All other subdirectories of the context, including the root directory, will contain normal files as HTML pages, style sheets, images, or JSP pages.

1.5. Adding a Servlet to a Web Application

A servlet is essentially a Java class implementing the *Servlet* interface. However, after compiling its sources and appropriately copying its class file in the WEB-INF/classes subdirectory, to make it available as a servlet resource, you must configure its features through a file called **web application deployment descriptor**. This file, named web.xml, must be placed in the WEB-INF subdirectory of the context.

A simple example of a descriptor is shown here.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee https://jakarta.ee/xml/ns/jak
         version="6.0">
 <display-name>Progetto IW</display-name>
 <description>Progetto X</description>
 <servlet>
  <servlet-name>Servlet1</servlet-name>
  <description>This servlet implements operation Y</description>
  <servlet-class>org.iw.project.class1</servlet-class>
 </servlet>
 <servlet-mapping>
  <servlet-name>Servlet1</servlet-name>
  <url-pattern>/operation1</url-pattern>
 </servlet-mapping>
 <session-config>
  <session-timeout>30</session-timeout>
 </session-config>
</web-app>
```

Each servlet is configured in a separate <**servlet**> element. The <**servlet-class**> element contains the full name of the class that implements the servlet.

Each servlet must be mapped to a URL using a <**servlet-mapping**>. The specified <**url-pattern**> will compose the servlet URL as follows:

```
http://[server address]/[context]/[url pattern]
```

1.6. The Servlet Base Classes

The base for a servlet implementation is the **Servlet** interface, which is implemented by a set of base classes like **HttpServlet**. All the servlets will be derived (*extends*) from this abstract class.

The other two base classes for the creation of a servlet are **ServletRequest** and **ServletResponse**.

An instance of **ServletRequest** is passed from the context to the servlet when it is invoked, and contains all information related to the request: these include, for example, GET and POST parameters sent by the client, the server environment variables, *headers* and *payload* of the HTTP request.

An instance of **ServletResponse** is passed to the servlet in order to return some content to the client. The methods of this class allow one to write to a *stream* which is then sent to the client, modify the HTTP response *headers*, etc..

1.7. The Servlet Lifecycle

The lifecycle of a servlet is marked by a sequence of calls made by container to particular methods of the Servlet interface.

- **Initialization**. When the container loads the servlet, it calls its `init` method. Typically this method is used to establish connections to databases and prepare the context for subsequent requests. Depending on the context and/or container policies, the servlet can be loaded immediately when the server starts, or after the first request, etc..
- **Service**. The client requests are handled by the container thorough calls to the `service` method. Concurrent requests correspond to executions of this method in separate threads. The implementation should therefore be thread safe. The service method receives user requests in the form of a ServletRequest and sends the response through a ServletResponse.
- **Finalization**. When the container wants to remove/deactivate the servlet, it calls its `destroy` method. This method is usually used to close database connections, or dispose other persistent resources activated by the `init` method.

The **HttpServlet** class specializes these methods for the HTTP communication. In particular, it contains two methods `doGet` and `doPost`, corresponding to the two most common HTTP verbs. The `service` method of HttpServlet class automatically redirects the client requests to the appropriate method.

1.8. Writing a Servlet Class

To write a simple servlet, you must create a class that extends **jakarta.servlet.http.HttpServlet** (or *javax.servlet.http.HttpServlet* with Java EE).

The servlet logic is coded in the methods corresponding to the HTTP verbs.

- `doGet` is called in response to GET and HEAD requests
- `doPost` is called in response to POST requests
- `doPut` is called in response to PUT requests
- `doDelete` is called in response to DELETE requests

All these methods have the same *signature*: they take a pair *(HttpServletRequest, HttpServletResponse)* and return *void*.

The **HttpServletRequest** and **HttpServletResponse** classes are specializations of ServletRequest and ServletResponse specific to the HTTP protocol.

The HttpServlet class provides a default implementation of all these methods, which only generates an error 400 (BAD REQUEST)

The servlet class contains other methods, such as `getServletContext`, which can be used to read a lot of information from the execution context of the servlet itself.

To compile a servlet, the package **jakarta.servlet (or javax.servlet with Java EE)** must be included in the **CLASSPATH**. A copy of this library, called *servlet-api.jar* is present in the common/lib directory of

Tomcat.

Example

```
package org.iw.project;

import jakarta.servlet.*;
import jakarta.servlet.http.*;

public class class1 extends HttpServlet {
 public void doGet(HttpServletRequest in, HttpServletResponse out) {
 //…
 }
}
```

A servlet class extends the basic jakarta.servlet.http.HttpServlet

For HTTP requests, the programmer should overwrite the appropriate methods: in this example, the `doGet` method is called to handle HTTP GET requests.

> *See the sample applications: Java_WebApp_Base_T10, Java_WebApp_Base_T9*

Example with deployment annotations

```
package org.iw.project;

import jakarta.servlet.*;
import jakarta.servlet.http.*;

@WebServlet(name = "Servlet1", urlPatterns = {"/operation1"})
public class class1 extends HttpServlet {
 public void doGet(HttpServletRequest in, HttpServletResponse out) {
 //…
 }
}
```

You can also find annotated servlets as in this example.

In this case, the annotation replaces the corresponding *servlet* and *servlet-mapping* elements in the web.xml file, which can then be omitted (see the deployment descriptor example in the previous slides).

1.9. Providing information on a Servlet

It is possible, although not required, to provide information about servlets that can be used by the container.

The information may include, for example the servlet author and the version number.

For this purpose it is sufficient to override the `getServletInfo()` method on the Servlet interface, which returns *null* by default.

The method takes no arguments and returns a string.

Example

```
package org.iw.project;

import jakarta.servlet.*;
import jakarta.servlet.http.*;

public class class1 extends HttpServlet {

 public String getServletInfo() {
   return "Example servlet, version 1.0";
 }

 public void doGet(HttpServletRequest in, HttpServletResponse out) {
 //…
 }
}
```

The string returned by `getServletInfo` the container will be used to provide information about the servlet.

1.10. Initializing and Finalizing a Servlet

The servlet initialization is accomplished in its `init` method, which has *ServletConfig* object as a parameter.

The first thing you should do is call the method `super.init()` passing it the ServletConfig parameter.

Then you can perform all the necessary initialization code, possibly by initializing the class fields with data that will be used by the service methods.

If the servlet has external initialization parameters (which are specified in a container-dependent way), you can read them through the HttpServlet method `getInitParameter`. This method takes as argument the name of the parameter and returns a string.

If the initialization has problems, you can throw a *ServletException* to report it to the container.

The servlet finalization is accomplished in its `destroy` method.

You must override this method only if there are things that you should do before the destruction of the servlet.

Example

```java
package org.iw.project;

import jakarta.servlet.*;
import jakarta.servlet.http.*;
I
public class class1 extends HttpServlet {
 private int parameter1;

 public void init(ServletConfig c)
  throws ServletException {
  super.init(c);
  parameter1 = 1;
 }

 public String getServletInfo() {
   return "Example servlet, version 1.0";
 }

 public void doGet(HttpServletRequest in, HttpServletResponse out) {
 //…
 }
}
```

The `init` method, after calling the same method of the super class, proceeds with the servlet initialization.

If there are initialization problems, it throws a ServletException.

1.11. Writing the Response

The **HttpServletResponse** object is provided to all the service methods and allows to build the reply to be sent to the client.

- The method `setContentType(String)` is used to declare the return type (such as "text/xml").
- The method `SetHeader(String, String)` allows one to add further *headers* to the request.
- The method `sendRedirect(String)` allows to redirect the browser to a new URL.
- The methods `getWriter()` and `getOutputStream()` allow to open a channel, text or binary, to write the contents of the response. It should be called after setting the *content type* and writing any other *header*.

Other methods allow you to manage, for example, cookies.

Example - html

```
package org.iw.project;

import jakarta.servlet.*;
import jakarta.servlet.http.*;
import java.io.*;
I
public class class1 extends HttpServlet {
 //…
 public void doGet(HttpServletRequest in, HttpServletResponse out) throws IOException {
  out.setContentType("text/xml");
  try (PrintWriter w = response.getWriter()) {
    w.write("foobar");
  }
 }
}
```

In the most common case, where you must return HTML text to the client, it is sufficient to take the response **Writer** through the method `getWriter()` and use it to write all the text to be returned to the browser.

Any other parameter of the response (in this case, the type) must be set *before* opening the output channel.

> See the sample applications: Java_Example_Servlet, Java_Example_Servlet_Fwk

Example - binary

```
package org.iw.project;

import jakarta.servlet.*;
import jakarta.servlet.http.*;
import java.io.*;
I
public class class1 extends HttpServlet {
 //…
 public void doGet(HttpServletRequest in, HttpServletResponse out) throws IOException {
  out.setContentType("image/jpeg");
  out.setContentLength(16000);
  out.setHeader("Content-Disposition", "attachment; filename=\"image.jpg\"");
  try (OutputStream s = out.getOutputStream()) {
   byte[] buffer = new byte[1024];
   int read;
   while ((read = resource.read(buffer)) > 0) {
    s.write(buffer, 0, read);
   }
  }
 }
}
```

When a binary file needs to be returned to the client, for example to perform a download, it is instead necessary to take the response **OutputStream** through the `getOutputStream()` method and write the data to be sent to the browser on it. In the example, we copy the data taken from a generic *resource* to the stream.

Before starting this operation, however, it is always a good idea to set a some *headers* necessary for the correct management of the download by the browser:

- the **Content-Type** tells the browser the type of the binary resource, thus making it possible to perform specific actions such as opening the internal viewer if you are downloading an *application/pdf*.
  We use the generic type *application/octet-stream* to indicate *generic* binaries that the browser will only allow to be saved;
- the **Content-Length** tells the browser the actual length (in bytes) of the binary, thus allowing estimates of the time needed for downloading;
- the **Content-Disposition** tells the browser whether the downloaded resource is embedded (*inline*) or external (*attachment*) with respect to the one from which the download was started (typically an HTML page). In the second case, you should also specify the **file name**, which will be proposed to the user as the default name when saving the resource.

> See the example applications: Java_Example_Downloader, Java_Example_Imager

1.12. Reading the Request

The **HttpServletRequest** object which is supplied to all the service methods contains information about the client's request.

- The method `getParameter(String)` returns the value of a parameter included in the HTTP request. If the parameter can have more than one value, use `getParameterValues(String)`, which returns an array of all the values assigned to the given parameter.
- For GET requests, the method `getQueryString()` allows you to read the entire query string (not parsed).
- For POST requests, methods `getReader()` and `getInputStream()` allow you to open a stream (text or binary) and directly read the request payload.
- The method `getHeader()` allows one to read the contents of the HTTP request headers.
- The method `getSession()` is used to manage sessions (see below).

Other methods allow to manage authentication, cookies, etc..

Methods inherited from class **ServletRequest**, finally, allow one to read information such request address (`getRemoteAddr()`) or protocol (`getProtocol()`).

These methods are not directly valid when you use the *multipart/form-data* encoding (file uploads)!

Example

```
package org.iw.project;

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

public class class1 extends HttpServlet {
 //…
 public void doGet(HttpServletRequest in, HttpServletResponse out) {
   String p1 = in.getParameter("p1");
 }
 public void doPost(HttpServletRequest in, HttpServletResponse out) {
   try {
     Reader r = in.getReader();
     //read raw payload raw from r…
   } catch(Exception e) {
     e.printStackTrace();
   }
 }
}
```

The `doGet` and `doPost` methods can read the parsed request parameters by `getParameter()` and `getParameterValues()`.

- doGet can read the raw query string calling `getQueryString()`.
- doPost can read the payload of the message from the stream returned by `getReader()` (text) and `getInputStream()` (binary).

Multipart

If request is sent to the server in **multipart/form-data** encoding (typically used if the request also contains binary files) we need to appropriately configure the servlet receiving it, otherwise it will not be correctly decoded.

We need to insert, within the corresponding *servlet* element in the web.xml file (or as an annotation on the servlet class if you use this second modality to declare the servlets in your project), the **multipart-config** directive, through which we can also specify:

- The location where the files attached to the request will be temporarily downloaded (**location**)
- The maximum size of each file that can be attached to the request (**max-file-size**)
- The maximum overall request size (**max-request-size**)
- The maximum size of files that will be kept in memory and not saved on disk (in the directory above) (**file-size-threshold**)

All of the above parameters have reasonable defaults, but it is always recommended to configure at least the maximum size of files and requests according to the requirements of your application, to avoid possible attacks.

After this change, the servlet will be able to access the data sent using the `getParameter()` already seen above, while for the files it will be necessary to use `getPart()` and then work with the **Part** object thus obtained.

Multipart example: servlet configuration

```xml
<web-app>
 <display-name>Progetto IW</display-name>
 <description>Progetto X</description>
 <servlet>
  <servlet-name>Servlet1</servlet-name>
  <description>This servlet implements operation Y</description>
  <servlet-class>org.iw.project.class1</servlet-class>
 <multipart-config>
  <max-file-size>20848820</max-file-size>
  <max-request-size>418018841</max-request-size>
 </multipart-config>
 </servlet>
 <servlet-mapping>
  <servlet-name>Servlet1</servlet-name>
  <url-pattern>/operation1</url-pattern>
 </servlet-mapping>
…
</web-app>
```

To enable a specific servlet to receive **multipart** requests, we must add the *multipart-config* directive to its definition.

The **multipart-config** directive can be also used to specify:

- The location where the files attached to the request will be temporarily downloaded (**location**), the system temp directory by default.
- The maximum size of each file that can be attached to the request (**max-file-size**), unlimited by default.
- The maximum overall request size (**max-request-size**), unlimited by default.
- The maximum size of files that will be kept in memory and not saved on disk (in the directory above) (**file-size-threshold**), zero by default.

Multipart example: servlet code

```
public class class1 extends HttpServlet {
 //…
public void doPost(HttpServletRequest in, HttpServletResponse out) {

 try {
  //ensure the request is multipart
  if (request.getContentType() != null && request.getContentType().startsWith("multipart/form-
   //normal form field
   String p1 = request.getParameter("p1");
   //uploaded file
   Part f1 = request.getPart("f1");
   if (f1 != null) {
    String name = f1.getSubmittedFileName();
    String contentType = f1.getContentType();
    long fileSize = f1.getSize();
    //move uploaded file from temp dir
    //to web app repository (target)
    Files.copy(f1.getInputStream(), target, StandardCopyOption.REPLACE_EXISTING);
   }
  }
 } catch(Exception e) {
  e.printStackTrace();
 }
}
```

In a servlet configured to receive **multipart** requests, you can use the normal `getParameter()` to retrieve all the data except files.

The `getPart()` method takes as an argument the name of the field corresponding to the file and returns a **Part** object through which you can read various information about the file itself such as name (`getSubmittedFilename()`), length (`getSize()`), and type (`getContentType()`).

The file is temporarily written to disk and must be moved to an application-managed directory if you need to keep it.

To this aim, it is useful to read the file as a stream (`getInputStream()`).

> See the sample applications: Java_Example_Servlet_Multipart_7, Java_Example_Uploader

2. Sessions

2.1. Introduction to Sessions

The concept of session is widely used in server-side programming. A session **associates state information to user requests**, allowing to circumvent the *stateless* feature of HTTP.

Typically, sessions are associated to a **unique identifier** *(session identifier)* that is assigned to the user when he/she enters the web application (for example, after the login), then is sent together with each subsequent HTTP request, and finally it is invalidated when the user logs out or after a certain period of inactivity.

The session identifier must be unique, and is usually generated with random algorithms based on the current date/time.

To send the session identifier inside each request, we can typically use two approaches:

- **Cookies**: Cookies are strings sent from the server to the browser. The browser automatically re-transmits to the server the associated cookies together with each request, so there's no need for specific client-side code. The so-called *session cookies* are deleted from the browser when it closes, and contain the session identifier.
  Cookies are the easiest and widely adopted solution, but may be affected by the security policies of browsers (such as disabling cookies).
- **URL rewriting**: session identifiers are written in the URL, as part of the path or, more commonly, as a GET parameter.
  This is the most generic and compatible solution, but requires the dynamic generation of all the website pages (each internal site link must be rewritten by introducing the current session identifier).

2.2. Managing Sessions with Cookies

In a servlet, creating and using a session through cookies is very simple. The session is managed by **HttpSession** objects. *Session variables* are simple strings that are associated with generic values of type Object.

First, we must take a reference to the session object by requesting it through the **HttpServletRequest** method `getSession(boolean)`.

If the parameter to *getSession* is true, a new session is created when there is not a valid one already active. Otherwise, the function may return null.

Calling this method can change the response headers, so it must be called before starting the output.

The HttpSession methods allow to manage the session:

- `isNew()` returns true if the session has been just created: in this case, usually, its state variables must be initialized.
- `getAttribute(String)` returns the object associated with the given name stored in the session.
- `setAttribute(String, Object)` associates with the specified name the object passed as the second argument. In practice, it creates or updates the state variable given by the first argument using the value contained in the second argument.
- `removeAttribute(String)` removes the given state variable.
- `invalidate()` closes the session and deletes all the state information associated with it.

Example 1

```java
package org.iw.project;

import jakarta.servlet.*;
import jakarta.servlet.http.*;
import java.io.*;
I
public class class1 extends HttpServlet {
 //…
 public void doGet(HttpServletRequest in, HttpServletResponse out) {
   HttpSession s = in.getSession(true);
   if (s.isNew()) s.setAttribute("pages",new Integer(1));
   int a = ((Integer)s.getAttribute("pages")).intValue();
   s.setAttribute("pages", new Integer(a+1));
   try {
     Writer w = out.getWriter();
     w.write("visited pages in this session: "+a);
   } catch(Exception e) {
     e.printStackTrace();
   }
 }
}
```

After each GET request to the servlet, if a session is not active, it is created and a variable called "pages" initialized to 1 (note the use of the Integer class) is placed inside it.

The number of pages visited during the session is then incremented and printed.

> See the sample applications: Java_Example_Servlet_Sessions

Example 2

```java
package org.iw.project;

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

public class class1 extends HttpServlet {
 //…
 public void doGet(HttpServletRequest in, HttpServletResponse out) {
   HttpSession s = in.getSession(true);
   s.setAttribute("user", in.getParameter("username"));
 }
}
```

This simple servlet shows how to save in the session the value of the "username" parameter taken from the request (probably from a form).

This is a typical operation performed at the end of a login process, to keep track of the user associated to the current session.

> See the sample applications: Java_Example_Login, Java_Example_Login_Middleware

2.3. Managing Sessions with URLs

Servlets also have a semi-automated system to manage sessions via URL rewriting.

In practice, in addition to the session management/creation/use code described in the previous slides, all you need is to transform any url that points to a resource in the same application using the method `encodeUrl(String)` of the object **HttpServletResponse**.

The method encodeURL determines whether you must put the session identifier in the URL: if cookies are available, the URL is not altered.

Example

```
package org.iw.project;

import jakarta.servlet.*;
import jakarta.servlet.http.*;
import java.io.*;
I
public class class1 extends HttpServlet {
 //…
 public void doGet(HttpServletRequest in, HttpServletResponse out) {
   HttpSession s = in.getSession(true);
   try {
     Writer w = out.getWriter();
     w.write(  out.encodeUrl(in.getServletPath())   );
   } catch(Exception e) {
     e.printStackTrace();
   }
 }
}
```

In this example, a session is created (if necessary) and the current servlet's URL, rewritten by `encodeURL()` to include the session identifier, is printed on the page.

If the browser supports cookies, the URL will not change.

3. Databases and Web Applications

3.1. Java and DBMS: the JDBC

One of the most common operations in a web application is the **management of data stored in a database**.

The data access in Java is done using the package **JDBC** *(Java DataBase Connectivity)*. A typical use of the JDBC classes follows the following steps:

- Make the **JDBC driver** for the DBMS in use available in the Java classpath.
- Load the driver by making a reference to the class that implements it using `Class.forName` method
- Proceed with the creation of a **Connenction** object using the static method `getConnetion` of the **DriverManager** class.
  The three parameters of the method are **the username and password** used to access the DBMS and the **JDBC connection string**, which specifies the address of the DBMS and the database to select: this string has a format that varies depending on the DBMS in use.
- Create a **Statement** object on the connection, using the method `createStatement`.
- Send an SQL query, as a string, to the DBMS through the created **Statement** and its method `executeQuery`.
  The returned object, of **ResultSet** type, allows to browse the query results.
  To send a query that returns no results, such as an insert or update statement, use the `executeUpdate` method. In this case, the return value is an integer representing the number of records processed by the query.
- Once the results have been processed, free up the space reserved for them by calling the `close` method of **Statement**.
- Finally, when the database is no more needed, close the corresponding connection by calling the `close` method of the **Connection**.

All JDBC instructions, in case of error, raise exceptions derived from **SQLException**.

Example

```
import java.sql.*;

Class.forName ("com.mysql.cj.jdbc.Driver");

Connection con = DriverManager.getConnection("jdbc:mysql://localhost/webdb?connectionTimeZone=

Statement stmt1 = con.createStatement();
ResultSet rs = stmt1.executeQuery("SELECT ** FROM test");
…
rs.close();
stmt1.close();

Statement stmt2 = con.createStatement();
int rc = stmt.executeUpdate("DELETE FROM test");
stmt2.close();

con.close();
```

This example creates a connection to a *MySQL* database.

The JDBC driver class is *com.mysql.cj.jdbc.Driver*. Note: if you use a MySQL driver version prior to 8, the class name is *com.mysql.jdbc.Driver*. Warning: many servers have pre-installed drivers for common DBMS. However, they may not have the latest version, especially in the case of the MySQL driver version 8. In this case, add the driver to your application!

The connection string specifies the DBMS type (*mysql*) the DBMS listening point (*localhost*), and the database to select (*webdb*). It can also include other parameters specified as a *query string*.

For the MySQl driver, since version 8, it is useful to use the parameters shown here to align the JDBC *timezone* with that of the server.

The connection also takes the username and password of the user to authenticate to the DBMS.

First a selection query is executed via `executeQuery` and then a delete query through `executeUpdate`.

The ResultSet

Through the **ResultSet** returned by the `executeQuery` method it is possible to read the columns of each record returned by a select query.

The records must be read one at a time. At any time, the **ResultSet** points (via a *cursor*) to one of the records returned (*current record*).

The values of the various fields of the current record can be read using the methods `GetX(column_name)`, where *X* is the **Java type** to extract (for example, `getString`, `getInt`,...) and *column_name* is the name of the field of the record to read.

To move the cursor to the next record in the **RecordSet**, we use the `next` method. The method returns *false* when the records are ended.

ResultSet Example

```
import java.sql.*;

Class.forName ("com.mysql.cj.jdbc.Driver");

Connection con = DriverManager.getConnection("jdbc:mysql://localhost/webdb?connectionTimeZone=

Statement stmt1 = con.createStatement();

ResultSet rs = stmt1.executeQuery("SELECT ** FROM test");

while (rs.next()) {
 System.out.println("nome = "+ rs.getString("nome"));
}

rs.close();
stmt1.close();
con.close();
```

In this example, we use a *while* loop to iterate through the results of a select query.

For each record we print the value of the *name* field, of string type.

Limitations of the standard usage pattern

In a *data-intensive* web application, where there are often many concurrent database accesses (many users may connect to the application simultaneously), the "standard" JDBC usage pattern presents considerable problems.

In fact, **opening a database connection is usually a complex process**, since it requires

- Loading drivers
- Connecting to the Database Server
- Authentication

We need to limit the overhead due to these operations, to make web applications as fast as possible.

3.2. Connection Pooling

Connection pooling is a technique that allows to **simplify the procedures needed to open and close JDBC connections** using a **connections cache**, called the *connection pool*.

The pool maintains a set of connections to a database **already opened and initialized**.

When the application wants to connect, **it takes a "ready" connection from the pool**, and operates on it.

When the application closes the connection, **it actually remains open and is returned to the pool**, waiting to be reused for other requests.

The pool is initially filled with a certain number of connections. However, if the pool is empty (i.e., all its connections are in use) and new connections are requested, they are automatically created "on the fly", enlarging the pool.

The connections left unused in the pool for too long can be closed automatically to free the corresponding DBMS resources.

Application server support

Connection pooling support is built-in the most recent versions of JDBC, but it have to be implemented in third-party software (just like a JDBC driver).

All the application servers provide an implementation of the connection pooling, and there are also external libraries used to add connection pooling support to any (non web) application.

Note: **Each application server provides proprietary systems to configure connection pooling**. We will see in particular how to use connection pooling with Tomcat.

In Tomcat

To configure connection pooling in Tomcat, we proceed as follows:

- **First, we configure the database connection in the web application context**, modifying the server.xml (global server configuration) or, better, the context.xml (specific application configuration).
- We add to the deployment descriptor (web.xml) a reference to the connection, which thus becomes **an application resource with type DataSource**.
- In the code, we takes a reference to the DataSource using the standard *Java Naming and Directory Interface* (JNDI).
- We create a normal JDBC connection through the DataSource.
- As usual, we close the connection when we don't need it anymore.

Application context configuration (context.xml) for Tomcat

```
<Context path="/Esempio_Database_Pooling">
 <Resource
  name="jdbc/webdb2"
  type="javax.sql.DataSource"
  auth="Container"
  driverClassName="com.mysql.cj.jdbc.Driver"
  url="jdbc:mysql://localhost/webdb?connectionTimeZone=LOCAL&amp;forceConnectionTimeZoneToSess
  username="website"
  password="webpass"
  maxActive="10"
  maxIdle="5"
  maxWait="10000"
/>
</Context>
```

The connection is configured though a **Resource** element, which contains all its features, including the driver, username, password and connection string.

The *type* must be specified as a **javax.sql.DataSource**. The *auth* attribute is set to "container".

The *maxActive*, *maxIdle* and *maxWait* attributes are used to size the pool, indicating:

- The highest number of connection that the pool will contain
- How many unused connections are allowed in the pool
- How long to wait for a new connection to become available

Warning: The JDBC driver must be copied in the lib directory of Tomcat. Simply copying it in the application WEB-INF/lib (as part of the deployment), will be it invisible to the class loader used for pooling!

Warning: the class name and the structure of the connection string may change depending on the version of the driver used by Tomcat (see previous examples)

Deployment descriptor (web.xml) configuration

```
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/
…
<resource-ref>
  <res-ref-name>jdbc/webdb2</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
…
</web-app>
```

In the deployment descriptor we add a reference to the JNDI resource with a **resource-ref** element.

The attributes *res-type* and *res-auth* reflect the *type* and *res* ones written in the **Resource** definition

res-sharing-scope should generally be set to "Shareable"

The *res-ref-name* attribute specifies the JNDI name used to access the resource in the code. The resources of type DataSource conventionally have a name that starts with "jdbc/".

Java code

```
try {
 //get a reference to the naming context
  InitialContext ctx = new InitialContext();
  //and from it a DataSource reference
  DataSource ds = (DataSource) ctx.lookup("java:comp/env/jdbc/webdb2");
  //database connection
  connection = ds.getConnection();
 //…use the connection…
} catch (NamingException ex) {
 //exception raised if the requested resource does not exist
} catch (SQLException ex) {
 //JDBC general exception
} finally {
 //the connection MUST be always closed!
  try { connection.close(); } catch (SQLException ex) {}
}
```

We first create a *JNDI naming context* (**InitialContext**)

Then we lookup the resource in the context. Note that the prefix "java:comp/env/" must be added to the resource name configured in the deployment descriptor. It can be useful to store such complete name in a web application initialization parameter.

We cast the returned object to the actual type of resource (**DataSource**)

Finally, we create the JDBC **Connection** using the method `getConnection()` of the DataSource.

After working on the connection, we close it as usual, and it will be returned to the pool. **Warning: if the connection is not closed, it won't return in the pool!**

Resource Injection

```
class DatabaseService {
 @Resource(name ="java:comp/env/jdbc/webdb2")
 private DataSource ds;
 //…
 public void dbMethod() {
  try {
   connection = ds.getConnection();
   //…use the connection…
  } catch (SQLException ex) {
   //JDBC general exception
  } finally {
    //the connection MUST be always closed!
    try { connection.close(); } catch (SQLException ex) {}
  }
 }
}
```

Resource injection allows to skip the complex resources lookup process: indeed, Java itself will inject a reference to the *DataSource* in a user variable.

The injection is usually performed on a class field, which **must have the correct type** (*DataSource*).

To perform injection, the field declaration must be preceded by the `@Resource` annotation, with the *name* parameter equal to the name of the resource to be injected.

> See the sample applications: Java_Example_Servlet_Database, Java_Example_Newspaper_DAO

4. Advanced Topics

4.1. ServletContextListener

A *context istener*, can be useful to perform any "global application initialization" (i.e., not specific to a particular servlet).

Objects that implement the interface **ServletContextListener** can be declared in the deployment descriptor (web.xml) as web application listeners.

The two methods `contextInitialized` and `contextDestroyed` of these objects are called, respectively, when the web application starts and stops.

These methods may perform any operation and change the **ServletContext** that will be a then passed to all the servlets at runtime.

Example

```
public class ContextInitializer implements ServletContextListener {

 public void contextInitialized(ServletContextEvent sce) {
   //initialize some global (context) variable
   sce.getServletContext().setAttribute("appID", 1);
 }

 public void contextDestroyed(ServletContextEvent sce) {

 }
}
```

```
<listener>
 <listener-class>it.univaq.f4i.iw.examples.ContextInitializer</listener-class>
</listener>
```

This context listener initialize some context variables when the web application starts, and stores them as ServletContext attributes.

Servlets can simply access them with an instruction like `getServletContext().getAttribute("appID")`

To activate the context listener, we simply **add the snippet below, which specifies the class name, to the web.xml**.

4.2. Filter

A *filter* can be used to **modify "on the fly" the input** (*HTTPServletRequest*) **and output** (*HTTPServletResponse*) data of a servlet.

Objects that implement the **Filter** interface can be associated to a web application by simply declaring them in the deployment descriptor (web.xml).

The two methods `init` and `destroy` of these objects are called, respectively, when the web application starts and stops.

The `doFilter` method is invoked **for each request** to the servlets the filter is configured for, and receives the request and response objects and a *FilterChain*.

Filters can **change the request/response objects** with custom implementations to control the input/output.

Filters must **call the doFilter method of the FilterChain**.

Example

```
public class EmailObfuscatorFilter implements Filter {
 private FilterConfig config = null;
 public void init(FilterConfig filterConfig) throws ServletException {
  this.config = filterConfig;
 }

 public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) thr
  chain.doFilter(request, response);
 }

 public void destroy() {
  config = null;
 }
}
```

```
<filter>
 <filter-name>emailfilter</filter-name>
 <filter-class>EmailObfuscatorFilter</filter-class>
</filter>
<filter-mapping>
 <filter-name>emailfilter</filter-name>
 <url-pattern>**</url-pattern>
 <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

This filters does nothing: it simply calls the `doFilter` method on the *FilterChain*.

To activate the filter, we **add the snippet below, which specifies the filter class name and the associated url patterns, to the web.xml**.

> See the sample applications: Java_Example_Emailfilter

5. References

Servlet API: Java EE 8 https://javaee.github.io/javaee-spec/javadocs/

Servlet API: Jakarta EE 9 https://jakarta.ee/specifications/platform/9/apidocs/jakarta/servlet/package-summary.html

Servlet Tutorial (legacy) https://docs.oracle.com/javaee/7/tutorial/servlets.htm

JDBC Tutorial http://docs.oracle.com/javase/tutorial/jdbc

Apache Tomcat https://tomcat.apache.org

6. Code Examples

The main code samples explained or developed during the classes are outlined below. These examples are all available in the GitHub, at the address https://github.com/orgs/WebEngineering-Univaq, and are a *key component* of the classes itself, since they illustrate the practical use of the concepts presented during lectures and reported in this documentation (where, when possible, references to these examples can also be found).

The list below may not always be up to date: in the repository you may often find useful new examples that have just been developed.

Template projects

- Java_WebApp_Base_T10
  Base project for a Java web application deployed on Tomcat 10
- Java_WebApp_Base_T9
  Base project for a Java web application deployed on Tomcat 9

Base servlets

- Java_Example_Servlet
  Base Java servlet example
- Java_Example_Servlet_Fwk
  The base Java servlet example rewritten using the course common utility framework

Response generation

- Java_Example_Downloader
  Example of binary data download with Java Servlets
- Java_Example_Imager
  Dynamic image generation with Java servlets

Request handling

- Java_Example_Post_Redirect_Get
  Safer POST submissions with the P-R-G pattern
- Java_Example_Servlet_Multipart_7
  Multipart request decoding with Java servlets
- Java_Example_Uploader
  A simple file repository with Java servlets

Sessions

- Java_Example_Servlet_Sessions

Basic session management with Java servlets

- Java_Example_Login
  Detailed login/logout process and session safety management with Java servlets
- Java_Example_Login_Middleware
  Login and session management with Java servlets and filters

Database (Model)

- Java_Example_Servlet_Database
  Basic database usage procedures in Java servlets

Templates (View)

- Java_Example_Templates
  Basic examples for the Freemarker template engine used in a servlet
- Java_Example_Templates_Fwk
  The Freemarker template engine used inside a simple rendering framework

Advanced topics

- Java_Example_Ajax_Pager_Async
  Server and client-side paging with javascript and Java servlets
- Java_Example_Emailfilter
  Using filters to mask email addresses in application output

Complete Web Applications

- Java_Example_Newspaper_DAO
  The Newspaper example, showing the full framework developed in the course

---

Title: Java Servlets
Author: Giuseppe Della Penna, *University of L'Aquila*
Version: 20250402
