Servlets 3 Fundamentals - Course course

Servlets 3

This course covers the fundamentals of the programming the Servlets according to the version 3.0 of the Servlets specification. After finishing this course student will be able to create base web applications using the Servlets 3.0 technology.

To understand the materials contained in this course you need to have solid understanding of the fundamentals of Java and Object Oriented Programming. Please note that you don't have to be an expert in all the Java SE modules and features.

After you finish that course you can learn more about advanced Java web technologies such as JSP, JSF, Vaadin, Struts or Spring MVC. This Servlet course is an excellent starting point for a Java programmer to learn Java web technologies.

The plan of the course is to:

- Prepare the Eclipse environment to exercise during the course.
- Set up a web container to deploy the exercises.
- Develop and improve the sample application (a book store).

1. Using a Web Server

As a web developer, starting and using a web server is a recurring task of your job and you probably do that more than ten times a day. Once you are comfortable with that task, you can learn writing and testing servlets.

In this lesson, you will learn how to inspect HTTP headers and write an HTML file. Then you will setup an IDE and a web server. Finally you will create some business classes to be used by your servlets in the next lessons.

1.1. Servlets, JSPs and Java EE

This topic present a brief overview of Java EE and how the Servlet/JSP technology relates to it.

1.1.1. Java EE

Java Enterprise Edition is mainly composed of two parts:

1. Servlet and JSP (JavaServer Page)
2. EJB (Enterprise Java Bean)

Servlets and JSPs run inside a web container, they are used to make web applications.

EJBs run inside an EJB container, they are used for back-end business logic.

It is possible to use one without the other, or to combine both. In this course, you only create servlets and JSPs in a web container. They enable you to write web applications, with pages presenting your business data, typically extracted from a DB.

1.1.2. Servlets and JSPs

A servlet is a Java class that you write and that will be called when the user types the associated URL in the browser. That Java class controls which data to show and produces a page sent to the browser.

A JSP is a text file that you write and that produce a page sent to the browser, with dynamic parts, typically data from your DB. A servlets and a JSP are usually combined together to produce one page sent to the browser.

In the first lessons of this course, you will focus on creating servlets. You will develop a small book application presenting a list of books. You can click on a book and view it's detail. You can edit a book using a form.

In the last lessons of this course, you will create JSPs, that combined with your servlets, will improve the technical design of your application and make its development and maintenance easier.

Contrary to the diagram below, you use no DB in this course, because it is a completely separated aspect and you need to focus on the servlet/JSP technology.

1.2. HTML Basics

Your program in a Java Application server will generate HTML text. Most programmers are not expected to be HTML gurus, neither graphic design specialists. But all web programmers must know the basics of HTML. It will help you to structure the layout of the pages that your program will send back to the browser.

For example, when you show an input form to a user who registers your application. You requires him to fill in his first and last names, his e-mail address, his password, and so on. You need to be able to produce an HTML form, but also to nicely display it with a label in front or on top of each field.

You do not need to be an HTML expert to follow this course. However, a minimum will do help you, as knowing how to define and use hyperlinks.

1.2.1. What is HTML?

The HyperText Markup Language describes the layout of your web page. It’s called a markup language, because tags (and attributes) are used to give information about the text between them. HTML is a markup language used to display formatted text.

The HTML specification defines various tags also called markups.

- Physical markups describe how text should be formatted : `<b>` `<h1>` …
- Logical markups define the structure of the text : `<head>` `<body>` …

The following HTML source,

Source

```
<!doctype html>
<html>
   <head>
       <title>Hello World example</title>
   </head>

   <body>
       <i>Hello World</i>

       <hr/>

       Hyperlinks to
       <a href="http://www.w3.org">W3C</a> and
       <a href="http://jcp.org">Java Community Process</a>.
   </body>
</html>
```

will be rendered as:

1.2.2. General Syntax

HTML files consist of text enclosed between a start tag (as `<b>`) and an end tag (as `</b>`). Some tags can contain attributes : `<body bgcolor=”white”>` HTML is non case-sensitive. Browsers ignore extra spaces, tabs, blank lines, new lines, …

Comments can be included in HTML source using the following syntax:

```
<!--This will not be displayed -->
```

1.2.3. Logical Markups

The most common logical markups are:

- `<html>`: delimits your html document.
- `<head>`: defines document-wide description elements as `<title>`.
- `<title>`: defines the title of the page that should be displayed in the caption of the browser window.
- `<body>`: content that will be displayed on the browser page (contrary to the `<head>` element).

1.2.4. Physical Markups

It is common to use these tags for simple formatting:

- `<b>`: bold
- `<i>`: italic
- `<u>`: underline

More advanced formatting is done through CSS files, which are out of scope of this course.

Glossary: CSS: XXXXXXXXXXX ADD a short CSS definition that your mother would understand XXXXXXXXXXXXX

During this server side programming course, we'll also need hyperlinks. Hyperlinks are links to other resources (typically html files). When you click an hyperlink, the browser jumps to the document pointed by the hyperlink. It allows you to navigate through the internet :

```
Hyperlinks to <a href="http://www.w3.org">W3C</a>
```

That tag displays the text "W3C" and makes it clickable. When clicked, the browser navigates to the URL value of the *href* attribute of the *a* tag.

1.2.5. Forms

We will also use forms. You can create HTML form, which will be used as an interface for the client to encode data. That data will be stored in the HTTP request and sent to the server :

Source

```
<!doctype html>
<html>
   <head>
       <title>Hello World example</title>
   </head>

   <body>
       <i>Hello World</i>

       <form method="GET" action="myservlet">
           Employee Name : <input type="text" name="employee">
           <input type="submit" value="ok">
       </form>
   </body>
</html>
```

The tag *form* has two important attributes:

- *method*: what kind of HTTP request is sent when the "ok" button is pressed (GET or POST)
- *action*: what servlet (what part of your code) should be activated when the "ok" button is pressed.

1.2.6. Activity: Create an HTML File

The goal of this exercise is to write a simple html file, to visualize it in a browser without a web application server.

1. Open Notepad, write a simple "Hello World" html file using the exemple given in this section
2. Save the file and double clic on it to open it with your favorite web browser. You should see this result:
3. Look at the location in the address bar of your browser. You can notice that the location is relative to the path where your file is located in your file system and not to your web application server as shown in the screenshot of section 1.2.1
4. look at the code source. Right click somewhere in the page displayed in you browser, in the menu you will find an option as "View source", the option name depending of your browser... You can compare the displayed source and the source in your file

1.2.7. Optional Activity: Create a Form

If you have more than one browser installed, it's interresting to look at the rendering differences beetween them, even for a simple code.

For that, add the form part to your html file using the exemple given in this topic. Open you file with different browser as for exemple Chrome, Firefox, Opera and IE...

You will see that the rendering of the button is most of the time different. Imagine the result of a complex webpage...

1.3. HTTP Basics

The server-side Java code that you will write is going to interact with a client browser. Having some insight on how that communication occurs is very useful to understand how the server and your code handles problems as passing parameters, identifying a session and storing cookies. In this topic, we will review HTTP, the communication protocol used by the client and the server. In the next lessons, we will explain how the web container uses that protocol to provide you high level features.

For example, your web Java application may provide a login feature. Users can login and the server remembers who they are. Such a mechanism is typically built over the notion of HttpServlet that will be explained in the next lesson. In order to understand that mechanism, it is required to know that HTTP is stateless.

1.3.1. What is HTTP?

HyperText Transfer Protocol (HTTP) defines the communication between a web browser and a web server :

- how a browser request is structured
- how a server response is structured

It is a transport mechanism.

Web browsers use the HTTP protocol to retrieve files from web servers, typically HTML files. HTTP can be used to

transmit any data that conforms to the Multipurpose Internet Mail Extension (MIME) specification.
Glossary: MIME: internet standard defining the type of data transferred through e-mail or through HTTP. That type can typically be text/html, or a graphics, audio and video. When a program sends content, MIME is a way to identify the nature of that content. It is like saying, "please look at the content I'm sending as an jpg image, and not as an mp3 sound stream."

There are many web server software products, such as Tomcat, Websphere, Glassfish. This course assumes the use of the *Tomcat* web server. A web server waits for requests from a browser. There are also many common web browsers; you can use any one you want to connect any standard web server.

Difference between HTTP & HTML:

- HTTP transports files from the server to the browser, often these files are text files formatted with HTML tags.
- HTML is a markup language that defines how a text should be formatted (bold, italic, font, size) for output.

1.3.2. Web Server – Web Client Interaction

A server computer offers services to other computers. A client computer uses the services of a server.
The typical web transaction is a client taking the initiative to send a request to a server; and the server sends a response to the client. They communicate using the HTTP protocol, often used to send HTML messages. So, the server's main role is to wait for HTTP requests from clients.

1.3.3. Communication Protocols

Http is the most commonly used protocol between a browser and a web server. There are other protocols at the same level for serving other purposes:

- FTP: File Transfer Protocol. It is session oriented (stateful): the client logins and initiates multiple file transfer operations. You can't view websites with FTP, but most of the time you will use FTP to upload your website on a server.
- SMTP: Simple Mail Transfer Protocol: transports a mail from the sender's machine to the target mail server and identifies the recipient's mail box within the mail server (with an e-mail address).
- Telnet: Used by virtual terminal software to get (green character oriented) screens to display from a server, and sends back the key pressed by the client user.

HTTP packets are transported over TCP/IP which is the network protocol of the Internet.

Below application level protocols as HTTP, FTP, SMTP and Telnet, we have the TCP/IP protocol stack to physically transfer the data packets from one machine to another. It includes a mechanism to identify machines (IP addresses). The data of an HTTP request is put inside a TCP packet, with a TCP header and, as content, the HTTP request. That TCP packet is put inside an IP packet, and so on.

GLOSSARY: TCP/IP: it's a suite of two protocols TCP (Transmission Control Protocol) / IP (Internet Protocol) used to connect hosts on the internet. IP (OSI Network Layer) provides a way to route datas to their destination. TCP (OSI Transport Layer) provides a way to exchange streams of data.

The TCP layer manages the port numbers to identify what program to reach on a machine. On the same computer, every program (process) that participates to TCP/IP communications gets assigned a port number to listen to. The default HTTP port is 80. Protocols other than HTTP have other defaults, such as port 21 for FTP.

1.3.4. HTTP is Stateless

HTTP is a stateless protocol, it supports only one request/response per connection. The client connects to the server by sending a HTTP request, receive a HTTP response, and then disconnects. Several connections to the same web server are thus independent from each others.

1.3.5. URL

A Unified Resource Locator (URL) can identify a resource that a HTTP client can fetch:

Domain : the server name is composed of a domain name (i.e. *w3.org*) and a host (machine) name (i.e. *www*).

In an URL, the server name may be followed by the TCP port number if port number other than the default is used: *http://www.w3.org:2001*. The default HTTP port is 80. Protocols other than HTTP have other defaults, such as port 21 for FTP.

This URL will be used often during this course:

```
http://localhost:8080/MyProject/index.html
```

- *localhost* is a generic name for your machine (the same as the browser).
- *8080* is a port often used during development.
- *MyProject* is your web application within the Tomcat web server, as we will explain later.
- *index.html* is the file that you request.

1.3.6. HTTP Requests

an HTTP request packet wraps the targeted URL and a message body plus some header information.

For example, to access this URL,

```
http://localhost:8080/MyProject/index.html
```

the *Chrome* browser sends that HTTP request:

Source

```
GET /MyProject/index.html HTTP/1.1
Host: localhost:8080
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.1 (KHTML, like Gec
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
```

The HTTP Request specifies the *HTTP method used*. The most common methods used are GET and POST. GET retrieves data from the server and POST sends data to the server. In this example the browser is trying to get the *index.html* file from the *MyProjects* directory.

The web server will dissect the request, determine which file is targeted and construct an HTTP Response.

1.3.7. HTTP Responses

The HTTP Response acts as an envelope containing the status of the response, the file requested and some additional information.

This is the response sent by Tomcat v7 (a Java web server that you will use in this course) to the request above:

Source

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Accept-Ranges: bytes
ETag: W/"0-1316864915299"
Last-Modified: Sat, 24 Sep 2011 11:48:35 GMT
Content-Type: text/html
Content-Length: 0
Date: Sat, 24 Sep 2011 11:49:12 GMT

<!doctype html>
<html>
   <head>
       <title>Hello World example</title>
   </head>

   <body>
       <i>Hello World</i>

       <hr/>

       Hyperlinks to
       <a href="http://www.w3.org">W3C</a> and
       <a href="http://jcp.org">
                                 Java Community Process</a>.
   </body>
</html>
```

The Status Code returned by the Web Server indicates whether the request was properly treated.

Some status codes :

- 200 : OK
- 403 : Forbidden
- 404 : Not Found
- 500 : Internal Server Error

In general:

- 1XX: information messages
- 2XX: success
- 3XX: redirection
- 4XX: webclient error e.g. not found or invalid credentials
- 5XX: webserver error

The *Content-Type* header line gives to the client (the web browser here) the MIME type of the response. The web browser reacts following its “knowledge” of this MIME type.

For example, the browser receiving *“Content-Type: text/plain”* prints the response as plain text, while the browser receiving *“Content-Type: text/html”* interprets and renders all the HTML tags in the response in order to print the entire text received with the proper formatting. Note that there are other types of MIME than text, such as “image/gif” or “audio/mpeg”.

The *Content-Length* header line gives to the client the size in bytes of the response received. It has to be the exact response's size. This size should be given especially if the response's content is binary (mpeg, gif...): one reason would be that this is the way your browser knows how many bytes it has to read and hence can show you a download progress percentage bar when you download a file. There are other reasons beyond the scope of this course. For text responses (plain or html), it is not a “must-have” but let's say it’s a good habit to take for the prior reason.

1.3.8. MIME

Mime types (**M**ultipurpose **I**nternet **M**ail) was originally a way to format non-ASCII text to be sent via email. They have been expanded to be used for sending non html text over Internet. This enables browsers to send and receive graphics, audio and video files via Internet. A mime type has the form type/subtype i.e. text/plain, text/html and can be found at the Content-Type header of the http protocol. When the browser receives “Content-Type: text/plain” prints the response as plain text, whereas when the browser receives “Content-Type: text/html” interprets and renders all the HTML tags in the response in order to print the entire text received with the proper formatting. Some examples of other common mime types are application/pdf for pdf files, application/vnd.ms-excel for excel documents, application/postscript for postscript files, "image/gif" for images, . In Java technology the Content-Type header is set by the setContentType method of the HttpResponse object, *response.setContentType("image/gif")*. If the browser does not support the MIME type of the HTTP response it returns "HTTP Error 406 Not acceptable".

For example, Note that there are other types of MIME than text, such as “image/gif” or “audio/mpeg”.

1.3.9. HTTP Methods

While the GET method is the most commonly used, the HTTP protocol contains others:

GET	Requests a resource, as for example a file. The URL may contain parameters, but they are limited in length: 2Kb
POST	Similar to GET: requests a resource. However, parameters are passed in the body and are not limited in length. POST is typically used with HTML forms for upload the data typed by the user, as we will see later.
HEAD	Requests information about a resource, usually to indicate if the server contains a newer version than the browser's cache.
PUT	Sends a resource to the server, similar to POST.
DELETE	Removes a resource from the server.
OPTIONS	Lists the available methods for a given resource. For example *is it deletable?*
TRACE	Sends an echo to the client. It's kind of like the TCP/IP ping utility.
CONNECT	Reserved for a further usage in the protocol.

In this course, we'll only explicitly use GET and POST.

HEAD and OPTIONS are more likely to be used transparently by your browser.

PUT and DELETE are used in *Restful* mechanisms which used in Ajax applications.

Glossary: **Ajax:** (Asynchronous Javascript and XML) is a way to construct web applications or dynamic websites using a combination of different technologies. The goal is to realise Rich Internet Application : applications that are more comfortable to use than traditional web applications. Ajax is based on built-in browser functionalities, it doesn't need to install a software framework to reach its goal unlike Adobe Flash, Silverlight, Java,...

1.3.10. Activity: Look at the Headers

As a little exercise, we are going to investigate the HTTP headers on an example site.

There are different tools to do this. Most browsers have integrated tools specific for developers.

- Chrome has a network tab when you press F12
- Firefox can use the FireBug plugin with Pagespeed(optional). This can be activated with F12.
  In the console tab > everything, you can click on the request and access parameters, headers etc.
- Firefox can also be used in combination with the Tamper Data plugin.
  Once installed, activate the plugin from the menu (Extra -> Tamper data).
- IE 8 has no integrated debug tool.
  Either you will have to install extra tools like Fiddler or Debugbar or install IE9. The developer tool for IE9 does now include a network tab with this information when you press F12

When even low level network analysis is required, you can install tools like Wireshark for network traffic en packet analysis.

Now install or activate these plugins and try to use them. Use http://www.example.com to investigate what happens. The first request is already interesting...

1.4. Install Eclipse and Tomcat

It is time to install a web server and make it run.

In this course, we use Tomcat. If you are comfortable with web containers, feel free to use any other one.

The setup is explained outside this course, in a special course containing common parts for various courses. Complete the following external chapter to install Tomcat and start it from Eclipse:

- Tomcat Setup with Eclipse - Common Chapter

then come back here, in the *Web Java Fundamentals* course.

1.4.1. Install

XXXX Section will be deleted, please move translations there:

http://knowledgeblackbelt.com/#CoursePage/13543899

1.4.2. Server Creation in Eclipse

XXXX Section will be deleted, please move translations there:

http://knowledgeblackbelt.com/#CoursePage/13543899

1.4.2.1. Exercise

XXXX Section will be deleted, please move translations there:

http://knowledgeblackbelt.com/#CoursePage/13543899

1.4.3. Products

1.5. Create the BBStore Project

You have installed and IDE and a web server and you are now ready to code.

A prerequisite to this course is being able to write simple Java SE programs. You will create some business classes before starting the web topics. If you are comfortable with Java SE, this activity should be done very quickly. Else, if it takes longer, it is a good opportunity to have some more Java programming experience and comfort before starting the web stuff.

A typical experienced Java programmer will take between 8 and 15 minutes to do it.

You will use these business classes through the course to have a realistic context for your servlets to display and manipulate data. In this topic, you will make these classes run inside a simple JVM. In the next lesson you will make them run inside the web container, with your servlets.

1.5.1. Functionalities

Our Web Application is a basic on-line bookstore. To build it, we need at least a Book class (which represents a book) and a BookList class (our books catalog).

The Book class contains only:

- four attributes:
  - String isbn
  - String title
  - String author
  - int price
- a single constructor accepting four parameters (one per attribute)
- one getter per attribute

The BookList class stores books in a HashMap (key=isbn, value=Book object)

- In its default constructor, create a few books and add them in the HashMap.
- Implement a public `List<Book> getBooks()` method which returns all books.
- Implement a public `Book getBook(String isbn)` method which returns the corresponding Book object, or null if the passed isbn code cannot be found.

1.5.2. Steps

1. In your BBStore project, create a com.example.bbstore.service package.
2. In that package, implement the Book class
3. In the same package, implement the BookList class
4. Create a Main class with a main() method: herein, you'll test that you can display all books to the console and that you can find a book by isbn.
5. Test your BookList class.

Hint: To write the getBooks() method, you may return a newly constructed List containing all HashMap values. This can be written in one line (see List and HashMap javadocs).

1.5.3. Global access to the books

To provide the global access point to the collection of our books, we will define the *UglyGlobalVar* class holding the static reference to the latter collection.

Source

```
public class UglyGlobalVar {

  private static BookList bookList = new BookList();

  public static BookList getBookList() {
     return bookList;
  }

}
```

Please note that this is only a helper for the exercise. In real world design, it is not recommended to hold the business data in the global static references. You will move the reference to BookList when you will have gained more knowledge about the Servlet API.

2. Writing Servlets

Now that we have Eclipse and Tomcat installed, we are ready to write servlet code.

A common task for a UI developer is to create master-detail interfaces. For example, the user gets a page with a list of clickable books. When he clicks on a book, he gets another page with the details of that book. If the books data is stored in a DB, we need to write code to get that data and present it as an html page. A servlet can achieve that.

In this lesson, you will create servlet classes, structure your project directories properly for Tomcat, handle request parameters and create links with parameters to enable the user navigate between your servlets.

2.1. Create a First Servlet

Servlets are the core technology of Java web programming. Even when you use a framework such as Struts, JSF, Spring MVC or Vaadin, servlets are used. They may not be be not written by you but provided by these frameworks.

You will create a servlet class for displaying the details of a specific book. You will learn the role of a deployment descriptor to configure your web server.

2.1.1. Running Inside a Web Container

Contrary to Java SE application, Java web applications have no *main* method. They run inside a *web container* which call your application every time a request arrives from a browser. Many products implement web containers, as for example Tomcat that you installed.

The web container provides interesting services for your application which will shield your application from some technical details. The web container:

- manages the network connections with the clients (browsers),
- understands the Http protocol and analyses requests for you,
- provides the request parameters in an easy format,
- provides a session mechanism to remember data (as the logged in user) between two requests of a specific browser,
- and many other services.

The Servlet deployment unit consists therefore from the two things:

- one or more classes extending javax.servlet.HttpServlet
- optionally a web.xml file (deployment descriptor). This happens if you decided not to use annotations introduced in the Servlets 3.0 specification.

Let's review these.

2.1.2. Servlet

A servlet is an entry point for your application on the server. When a request arrives, the web container calls a method of your servlet.

The simplest possible servlet extends javax.servlet.HttpServlet and is annotated with @WebServlet.

Source

```
package com.example.hello;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;


@WebServlet("/HelloWorldServlet")
public class HelloWorldServlet extends HttpServlet {

@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response) thr
       PrintWriter out = response.getWriter();
       out.println("Hello <b>World</b>!");
}

}
```

In the code above, we create our own servlet class. It extends HttpServlet. All Java EE compliant application servers will provide the classes from the servlet specification on the classpath of your application. HttpServlet defines the *doGet* method, and you override it in your class. It is called by the web container when a request arrives from a browser. The web container gives us two parameters:

- request
- response

In this hello world example, we only use *response*. We get a PrintWriter out of it, to have access to *println* methods. What we printed through *out* will be sent back to the browser, which will display "Hello World!".

On this screenshot, we pressed F12 in IE to make the bottom part appear, with the HTML structure of the page displayed on the top part.

2.1.3. URL Path

This servlet is reached with the following URL

Source

```
http://localhost:8080/HelloWorld/HelloWorldServlet
```

In this URL, localhost is the machine where the browser resides. Your target, the Tomcat server, runs on the same machine as your browser.

8080 is the default port number used by Tomcat when it is in development mode.

HelloWorld is the name of your web application within Tomcat. By default, it is the same as your Eclipse project name. You can modify it by righ-clicking on your server (in the Eclipse Servers view), and selecting open. In the view that just opened, select the *Modules* bottom tab. A *module* in this context is a web application. You could deploy multiple applications on the same server.

The last part of the URL is HelloWorldServlet. It is the URL Pattern of your servlet.

2.1.4. URL Pattern

The configuration of a Java web application includes the mapping between URLs and Servlets. You have given that information through the @WebServlet annotation:

```
@WebServlet("/HelloWorldServlet")
public class HelloWorldServlet extends HttpServlet { ... }
```

The parameter of @WebServlet is the URL pattern. It that string is found after the web application name in the URL, then the class com.example.hello.HelloWorldServlet gets the request for processing.

The urlPattern attribute is more verbose way to indicate it. It takes an array of patterns.

```
@WebServlet(urlPatterns={"/Foo", "/world/*"})
public class HelloWorldServlet extends HttpServlet { ... }
```

The configuration above, accepts all the following URLs:

```
http://localhost:8080/HelloWorld/Foo
http://localhost:8080/HelloWorld/world
http://localhost:8080/HelloWorld/world/apage
```

but NOT these URLs:

```
http://localhost:8080/HelloWorld/Foo/something
http://localhost:8080/HelloWorld/foo
```

2.1.5. Activity: Create Hello World Servlet

Create your first servlet. It might be especially easy if you use the Eclipse servlet creation functionnality.

If you do it manually, you will need to perform the following steps:

1. In Eclipse, create a dynamic web project and a corresponsing server, as explained in the setup topic.
2. In your project, create a servlet class. In Eclipse, you may use the servlet wizard: file > new > other > Web > Servlet.
3. Annotate that servlet with @WebServlet and some URL pattern
4. Fill the code in the doGet method, to renders "Hello World!" in the response.

5. (re-)start your server
6. Open your web browser to the servlet Url: http://localhost:8080/YourEclipseProjectName/YourUrlPattern

2.1.6. Deployement Descriptor

So far, we have used an annotation to specify configuration values (the URL pattern) for our web application. Any value that can be specified through annotations, can alternatively be specified through an external XML configuration file. That file has the role of *deployement descriptor*. It is always named *web.xml* and must be located in the WEB-INF directory of your web application.

NOTE: a valid directory structure for your web project has automatically been created by Eclipse. It includes the WEB-INF directory.

Since the v3 of the servlet specification, the deployement descriptor is optionnal. So far, we did not need it because we gave enough meta-data through annotations. In some cases, using the deployment descriptor is mandatory. For example, when you use a framework as Struts, JSF or Vaadin, they come with their own servlet. You need to configure that servlet, including an URL pattern of your choice. However, you are not going to change the source code of these frameworks to insert a @WebServlet annotation. The web.xml deployement descriptor enables you to provide that information externally.

Annotations and a deployment descriptor can be used together. For example, in the same project, one class can be defined as being a servlet through the @WebServlet annotation, and another class can be defined as being a servlet in the deployment descriptor.

2.1.7. Analyzing the Deployment Descriptor

If we remove the @WebServlet annotation from our HelloWorldServlet class, we need to create the following web.xml file to keep the same behavior.

Source

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/
        version="3.0">

   <servlet>
       <servlet-name>HelloWorldServlet</servlet-name>
       <servlet-class>com.example.hello.HelloWorldServlet</servlet-class>
   </servlet>
   <servlet-mapping>
       <servlet-name>HelloWorldServlet</servlet-name>
       <url-pattern>/HelloWorldServlet</url-pattern>
   </servlet-mapping>
</web-app>
```

You find at the first line the `<?xml ... ?>` header that specifies the version and the encoding for this xml file. In the second line, you din the `<web-app>` opening tag. Its attributes (xmlns=...) identifies the xml schema, which defines “grammar” to be used to “read” what elements are written after. Those schema declarations are mandatory and helps the web container to identify what version of the servlet specification your code is using. These first lines are above all xml features. In this course we focus on the elements contained between the `<web-app></web-app>` tags.

In the `<servlet>` element, we identify our servlet class by its fully qualified name, and we assign a symbolic name to it *HelloWorldServlet*. That symbolic name does not have to be the same as the class name, but it would be confusing to give another name.

The `<servlet-mapping>` element tells the URL pattern which should trigger the servlet. We associate it with the servlet symbolic name assigned to the servlet class.

2.1.8. Activity: Using the Deployement Descriptor

We start from our HelloWorld project with the annotated servlet, and create the web.xml file.

1. Re-test your project to display the hello world text in your web browser.
2. Remove the @WebServlet annotation on the HelloWorldServlet class.
3. Restart the server and check that the browser cannot access the servlet anymore.
4. Create the web.xml file under the existing WEB-INF directory. You can also ask Eclipse to generate it. In the Project Explorer view, one of the very first nodes of your project in the tree is *Deployement Descriptor: HelloWorld*. It is a symbolic "view" of your (non existing yet) web.xml. Right click on it and select *Generate Deployement Descriptor Stub*.

5. Adapt the deployment descriptor with working servlet and url mapping information.
6. Restart your server and test from the browser.

2.1.9. Content Type

The servlet may send any kind of content. It could be an image, or a raw text file, for example. In most cases, the response contains an html formatted text.

You specify it with through the HttpServletResponse class, and the browser will get this tip as an HTTP header, to know how to display the received data.

This is the body of your servlet's doGet() method, modified to set the content type.

Source

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<html>");
out.println("  <body>");
out.println("     Hello World!");
out.println("  </body>");
out.println("</html>");
```

2.1.10. Activity: Create BookDetailServlet

You will create a servlet that produces HTML dynamically, to display each attribute of the first book in the books list.

2.1.10.1. Steps

1. Create a com.example.book.servlet package
2. In that package, create a BookDetail class extending HttpServlet.
3. Implement the doGet() method.
4. Map your servlet to the "/detail" as URL pattern, either via an annotation or the web.xml file.
5. Deploy the application and test your servlet.

2.1.10.2. Hints

- To complete this exercise, you'll have to use the HttpServletResponse parameter to:
  - set the content type to "text/html"
  - get a PrintWriter object to which you'll send your HTML output.
- In web.xml, you'll have to use the `<servlet>`, `<servlet-name>`, `<servlet-class>`, `<servlet-mapping>` and `<url-pattern>` elements.

2.1.10.3. Optional

Use a table (with an invisible border) with 2 columns to structure the info (label, value). The HTML syntax for a table:

Source

```
<table>
   <tr>
       <td>col1row1</td><td>col2row1</td>
   </tr>
   <tr>
       <td>col1row2</td><td>col2row2</td>
   </tr>
</table>
```

2.2. Web Application

Web Java projects must comply structural constraints defined by the servlet specification. So far, you have enabled the Eclipse IDE to create a correct structure for you. In this topic, you review the directory structure of a compliant servlet application.

2.2.1. Directory Structure

Eclipse created that structure for your project; let's review it:

- **MyProject**: this is the root of your Eclipse project.
- **MyProject/src**: contains your source code (package sub-directories). In the Eclipse *Project Explorer* view, it appears in a *Java Resources: src* branch.
- **MyProject/build/classes**: contains your compiled code. Does appear in the *Navigator* view but not in the *Project Explorer* (hidden).
- **MyProject/WebContent**: root of your web application once deployed. Its structure is detailed below.

A web application needs to be contained in a specific structure, for the server to find the deployment descriptor, classes, libraries, etc. Eclipse will prepare everything to match the servlet specification under the *WebContent* directory. At runtime, only this structure (under WebContent) is deployed.
Below WebContent:

- **/**: the root directory of the web app. All files here are directly accessible by the client (through an URL), as the html file we've created to test Tomcat.
- **/WEB-INF**: optionally contains the web.xml deployemement descriptor. Eclipse also shows the file through a branch *Deployment Descriptor* in the root of your project in from the *Project Explorer* view. Contrary to the root, no file in WEB-INF is directly visible to the browser.
- **/WEB-INF/classes**: contains the compiled classes of your project. Note that in the special (but usual during development) case where you run your application from within Eclipse, a trick is used to tell Tomcat to look somewhere else for the compiled code, which explains why that directory may be empty. Eclipse is setting that automatically and transparently when you create a *dynamic web project*.
- **/WEB-INF/lib**: contains JAR files that the application depends upon. You typically put frameworks here, such as Hibernate and Spring. Note that you can include jars from any directory in your Eclipse project build path, and those will be available at compile time. But to be available at runtime, jar files must be in WEB-INF/lib. Once a jar file is in WEB-INF/lib Eclipse automatically adds it in your project's build path.

2.2.2. Eclipse Default Output Folder

If you use Eclipse, you may have noticed that the compiled code (.class files) are placed in *MyProject/build/classes*

This is the default for a freshly created Eclipse *Dynamic Web Project*. That directory is hidden in the project and package explorer views. Use the Navigator view to see the real file structure (or use a tool outside Eclipse as Windows file explorer). To display the Navigator view: Window (menu) > Show View > Navigator.

This configuration doesn't match the required directory structure of a JEE webapplication, where compiled classes should be in the directory /WEB-INF/classes (or in a jar-file in /WEB-INF/lib). The above configuration does work when you run the project on a server, since Eclipse changed some of Tomcat's settings to make work with the default eclipse layout.

We don't like this situation because:

- It violates the specification without any benefit, and it's easier when things are where they supposed to be.
- It can have unfortunate side effects for us, as we will see in the Spring integration chapter.

To fix the configuration, we will do something very easy: change Eclipse output compilation directory to the correct place: *MyProject/WebContent/WEB-INF/classes*. Do that change very simply: right click on your project > properties > Java Build Path > source > Default output folder. From there, set the value of the *Default output folder* field to *MyProject/WebContent/META-INF/classes*.

2.2.3. Deployment

For deploying a web app on the server, you can just copy the previous described directory structure (under WebContent), with the appropriate servlet class files and deployment descriptor containing the mapping for them (and optionally other things – will be seen later), in the appropriate directory of you web server. E.g. in Tomcat, this is the *Tomcat_root*\webapps directory.

Eclipse modifies that default. You can set that the server's setting in Eclipse: Servers view > Right-click on Toimcat > Open > Server Locations.

By default, Eclipse sets the Tomcat deployment directory to the following sub-directory of your workspace:

Source

```
.metadata\.plugins\org.eclipse.wst.server.core\tmp0
```

In the example below, *web* is the Eclipse workspace directory where resides our HelloWorld project's folder. As shown in the figure, that HelloWorld folder has been copied by Eclipse in the tomcat deployment directory.

However, instead of copying the entire directory structure, the standard way is to create a WAR file from it, which can be used for deployment. A WAR file is a JAR file for the web (with the correct directory structure inside). It is just a zipped version of your WebContent directory.

For the activities of this course, we let Eclipse deploy by copying the file in the *tmp0* sub-directory automatically.

2.3. Request Parameters

It is very common for servlets to receive parameters from the browser. For example, a servlet displaying the details of a book should know which book to display. The most straightforward way is to put an identifier of the book in the URL, as a request parameter.

In this topic you will enhance the BookDetailServlet to get the isbn of the book as parameter.

2.3.1. Retreiving Parameters

An Http request from the browser may contain parameters. The following URL contains two parameters: name and id.

```
http://localhost:8080/MyProject/MyServlet?name=John&id=123
```

The ? sign delimits the servlet name from the parameters. Multiple parameters are separated by the & sign.

A cruciual part of web applications is to create and handle these parameters. To retrieve them, the *HttpServletRequest* class contains a *getParameter(String)* method. You give the parameter's name and it returns the value as a String. In our example,

```
String value = request.getParameter("id");
```

value would receive the string "123". Note that the *request* variable is a parameter of your servlet *doGet/doPost* method.

NOTE: you cannot change value of the HTTP parameters - you can only read them using the Servlets API.

2.3.2. Activity: Pass a Parameter to HelloWorldServlet

Start from your project with the HelloWorldServlet and make it say hello to a name passed as parameter.

1. Copy your working HelloWorld project: right click on it in the Project Explorer View, and select copy. Right click in an empty area of the Project Explorer view and select paste. Change the name to HelloWorldParam.
2. Add the necessary line to the doGet() method, to retreive the parameter named "who" into a String local variable named *who*.
3. Still in the doGet() method, change the println statement to include the value of the *who* variable.
4. Start your server and test your servlet with your browser. Add a name add the end of the url: *?who=John*

Source

```
http://localhost:8080/HelloWorld/HelloWorldServlet?who=John
```

2.3.3. Activity: API for Parameter Quiz

The answers of the following quiz are in the Java EE JavaDoc. You may also experiment(code) to verify your answers.

1. The getParameter(String) method will return …………..if the given parameter does not exist.
2. If a parameter has multiple values (e.g. from a checkbox), what will be returned by the getParameter(String) method? ……………….
3. If the parameter has no value, the getParameter(String) method returns what? …………………………….
4. How can you use the result of the method getParameterMap() to retrieve a parameter called “name”? ………
5. Which method can you use to retrieve all the parameter names in this request? Having these, how can you test whether the parameter “name” is set?

Solutions:

1. null
2. The first value.
3. null
4. request.getParameterMap().get("name")
5. Iterate over the Enumaration request.getParameterNames():

```
boolean found = false;
for (String pName : request.getParameterNames()) {
   if ("name".equals(pName)) { found = true; break; }
}
```

But it's probably easier to use `request.getParameterMap().get("name") != null`

2.3.4. Activity: Pass a Parameter to BookDetailServlet

Our last BookDetailServlet always displays the same book. We will adapt it to get the isbn as parameter, and display the corresponding book.

1. Start from your project with *BookDetailServlet* displaying the first book of the map.
2. From within the *doGet()* method, get the parameter named *isbn*
3. Add an if statement to print in the response *isbn parameter missing* if the *isbn* is null, and exit the method.
4. Retreive the BookList instance.
5. Call *BookList.getBook()* with the *isbn* as parameter, to retrieve the book.
6. Print in the response stream, the *isbn*, *title* and *price*.
7. In case the book cannot be found, the message *Book not found!* should be printed.

2.3.5. GET or POST?

There are two different HTTP methods you can use for sending parameters to the server, namely GET and POST. However, there is an important difference between them. When using GET, the parameters are included in the URL (which is in the HTTP header), so they're visible in the browser's input bar. When using POST, they're included in the message body, having as consequence that you can include more data, but that the parameters are not bookmarkable.

Above, we have used the GET method to send parameters. We'll now use an html form and switch between GET and POST for experiment.

In the following html form fragment, the GET method is explicitely selected. It could have been POST instead.

```
<form method="GET" action="myservlet">
   Employee Name : <input type="text" name="employee">
   <input type="submit" value="ok">
</form>
```

A parameter named *employee* will contain the text typed by the user in the text field of the form. The *action* attribute will tell where to send the request when the user presses the ok button. *myservlet* will be appended to the url from the web app root to form:

```
http://localhost:8080/MyProject/myservlet?employee=John&submit=ok
```

2.3.6. Activity: Post the isbn

Your task will be to create the plain HTML page with input form. The latter form will enable you to test *BookDetailServlet* from the previous exercise.

- Create an html file in *WebContent* named *SearchForm.html*.
- Add a field to the form to fill in an *isbn* number, which will be passed to *BookDetailServlet* via a submit button. The forum field should be named *isbn*.
- When you use the GET method, your *BookDetailServlet* from the previous exercise needs no change to work. Try to switch between the GET and the POST method, and note the visible differences.

Hint: When a request comes with the POST method, your *doGet* method is not called, but *doPost* is.

2.4. Navigating Between Servlets

In the previous topic, you made your servlet process request parameters. The other aspect of the story is to send these parameters to your servlet.

In this topic, you create hyperlinks with parameters from a servlet, to another servlet, and you create a natural navigation experience for the end user. You create a new servlet displaying all the available books as clickable links. Each links enables to display the details of a specific book.

While this topic is short and not technically complex, it is key to understand web applications, and it is often underestimated by students.

2.4.1. Navigation

In your web app, you need a way to navigate from one component (HTML page, servlet) to the other. One thing you can use is including hyperlinks in your pages.

```
<a href="http://example.com">Example web site</a>
```

You can print this out from your servlet, and even construct the parameters for the href attribute dynamically. E.g.

Source
```
String idvalue = "345";
out.println("<a href='http://example.com?id=" + idvalue +"'>Example web site</a>")
```

Which produces:

```
<a href='http://example.com?id=345'>Example web site</a>
```

NOTE: The result uses single quotes to delimit the href attribute value. In html/xml, you can either use double or single quotes. But in Java, you must use double quotes for Strings. To prevent the confusion between the limit of the Java string and the limit of the href attribute value, you used single quotes for the attribute.

2.4.2. Activity: Create BookCatalogServlet

You will create a servlet, that will produce hyperlinks to the BookDetailServlet, with the correct parameters. It will act as a menu where the user can pick the book of his choice to see its details.

Write a new servlet named BookCatalogServlet to display all the books contained by BookList in an HTML `<table>`. If you are not comfortable to make a table, just list the books with not table around.

Each book title should be a hyperlink to the *BookDetailServlet*.

1. Start from the project with the BookDetailServlet accepting an isbn parameter. You may copy this project within Eclipse and rename it.
2. In the com.example.book.servlet package, create a class named *BookCatalogServlet* extending *HttpServlet*.
3. Don't forget to annotate it and associate an URL pattern to it.
4. In the BookCatalogServlet.doGet() method,
    1. Retreive a collection of all the books through BookList.
    2. Create a loop to iterate over that collection.
    3. Inside the loop, add an hyperlink to BookDetailServlet, with the current book's isbn in the url, and the title of the book as visible text.
5. Start the server and test your catalog servlet, including the navigation to BookDetailServlet.

3. Using Scopes

Using servlets, you can create pages to display information of your application.

Sometimes, you would like to store information in the web container. For example, after a user logs in, you want your server side code to retrieve his user object or user id, on every following request that user will send. It enables you, for example, to display his name in the corner of every page. But you don't want to mix that up with other users logging in you system.

In this lesson, you will select the appropriate scope to store objects, such as a logged in user. You will store data in these scopes and retrieve it.

3.1. Request

A request object has a short live in the container. It is instantiated by the container when an HTTP request arrives and it is discarded by the garbage collector after the corresponding HTTP response is sent. It would indeed be a bad design/practice for your application to keep references to requests after the response has been sent.

In a simple scenario, an HttpServletRequest instance is given as parameter to the doGet/doPost method of one servlet, and that's it. In a more complex scenario, a request could be passed to a few servlets before the corresponding HTTP response is sent. That kind of scenario will be covered in detail in the next lesson and in the JSP part of the course.

When one servlet forwards the request to another servlet, how do they share information? The HttpServletRequest object is a very convenient place for the first servlet to store information to be retrieved by the second servlet. The mechanism of attributes enables that. It works like a map with pairs of key (String) and value (Object).

In the code below, we attached a book to the request, with the attribute name "bookToDisplay". This code happens in the first servlet.

```
Book book = ....
request.setAttribute("bookToDisplay", book);
```

The second servlet will try to retrieve the value of that attribute:

```
Book dspBook = (Book) request.getAttribute("bookToDisplay");
```

Of course, both servlets need to know the attribute's name "bookToDisplay" which is typically stored in a constant.

The request scope is the shortest scope of the servlet specification. It only lasts the time of one request which is supposed to be short (else your web site visitors will wait for the page to display). It makes sense to use it (to attach attributes) when multiple servlets are involved in that request processing and need to communicate values. That is not yet the case of our book application.

3.2. Hidden Fields

When we need information to be remembered accross two requests, an hidden field is a common strategy. As for cookies, the information is stored on the client browser and sent back with the next request. Contrarily to cookies, that information is kept on the page itself and therefore is only sent from that page, not from other tabs of browser to the same site.

3.2.1. HTML Hidden Field

The system relies on the notion of *hidden* type for an HTML input. The form below contains 3 input fields:

- a text field to enable to user to write an employee name,
- a hidden field to store information but not visible to the user,
- a submit button labeled with "ok".

```
<form method="POST" action="EmployeeEditServlet">
   Employee Name : <input type="text" name="employee"/>
<input type="hidden" name="ssn" value="123-45-6789"/>
   <input type="submit" value="ok"/>
</form>
```

In a browser, the hidden field is invisible to the user:

3.2.2. Passing Information

That mechanism can be used to pass informations. This form enables to change the name of an employee. It has been generated by a servlet that knows which employee is concerned. Then the user gets the form and enters a new name. He presses the ok button and the browser sends a request to the EmployeeEditServlet with the new name. How can EmployeeEditServlet know which employee we are talking about? It gets a new name, but for which employee?

Here is the scenario that will enable the EmployeeEditServlet to know which employee is being edited.

1. The EmployeeShowFormServlet generates an HTML form for editing the employee having the social security number 123-45-6789. It places a hidden field named "ssn" with that number in the form.

   ```
   out.println("<input type='hidden' name='ssn' value='"+ employee.getSsn() + "
   ```

2. The browser displays the form and the user inputs a new name for the employee.
3. The user presses the ok button, and an HTTP request is sent to the EmployeeEditServlet, with the value of every field.
4. The EmployeeEditServlet reads the value of the parameter "ssn" and gets the concerned employee number.

   ```
   String ssn = request.getParameter("ssn");
   ```

   It can update the name of the corresponding employee in the DB.

3.3. Session

In one of the previous chapters, we discussed how you can store parameters in the request, and retrieve them afterwards to use them in e.g. another servlet. So a request carries crucial data, while servlet code knows how to find and use it. However, web servers have a short-time memory. As soon as they send you a response, they forget who you are. The next time you make a request, they don't recognize you. They don't remember what you've requested in the past, and they don't remember what they've sent you in response.

In this chapter, we will see another object, called the session, to make data available, longer than one request of a client.

Sometimes you need to keep conversational state with the client across multiple requests. A shopping cart wouldn't work if the client had to make all his choices and then checkout in a single request.

3.3.1. Definition

The HttpSession is a object managed by the container, that is unique for one interracting user. This object is kept by the web container accross requests. Its set/getAttribute() methods enables to you store/retreive any business information associated to the current user.

In this example, we associate the value "John" with the attribute name "user".

Source
```
HttpSession session = request.getSession();
session.setAttribute("user", "John");
```

Then we can retrieve the value of the attribute "user" which is a String in this case but could have been an instance of *User* for example.

Source
```
String nameOfUser = (String) session.getAttribute("user");
```

We probably execute the *setAttribute* in a servlet (in the servlet responsible of login for example), and execute *getAttribute* much later from another servlet (to display the logged user name, for example).

3.3.2. HttpSession Usage

The web container (Tomcat) will manage the HttpSession object for you. There you can store the conversational state across multiple requests. There is one instance for each browser open on a machine. For example, if you open two tabs to your site in FireFox, they will share the same HttpSession instance. If you open IE and FireFox on the same PC, there will be one separate instance of HttpSession for each of them on the server.

In fact, with FireFox there's one session for all windows. In IE there's a separate session for each IE window.

In a session, you can store attributes, which are Objects. These objects need to be serializable because sessions can be serialized to secondary storage or can be migrated to another cluster.

Note the difference with request parameters, which are Strings.

As you can check in the javadoc of the HttpSession interface, attributes can be added and retrieved by setAttribute(String name, Object value) and getAttribute(String name) methods. The last one returns an Object, so you have to cast the return variable to its specific type.

Some more details about getting the session:

- The getSession method always returns a session, because it creates a new one if no session exists. On the session you can call isNew() to see whether a pre-existing session is returned or not.
- If you don't want to create a new session and only want to retrieve an existing one, you can use getSession(false). This method returns null, or a pre-existing session.

Note: you must call *request.getSession()* before any content is sent to client, because this method can set HTTP response headers.

3.3.3. How Does it Work Inside?

Now, how does the container retrieve the sessions which is associated with a specific client? Well, what happens is that the container creates a unique session ID when the session gets created for a specific client. With each subsequent request, the client sends its session id to the server.

When a request arrives, the server needs to associate it with a session ID. There are three possible ways of doing this:

- Cookies
- URL rewriting
- SSL session information

This diagram shows a side-by-side example of the cookies and URL rewriting strategies, and the text below explains the 3 ways.

Click to download

3.3.3.1. Cookies

A cookie is a piece of information created by the server and sent to the browser. The browser remembers that information on its hard drive and sends that information with every subsequent request.

Using cookies is the most easy way, because the server does all the work for you:

- The first time a client requests a session, a cookie gets created and sent back in the response.
- The next getSession method call from that same client, will find the existing in the request cookie, and reuse that already created session for that client.

3.3.3.2. URL Rewriting

Not all clients accept cookies, so we need a mechanism to do session tracking without cookies.

URL rewriting is the solution: the idea is to paste the session id at the end of every URL that is send to the server. Instead of you having to paste it there manually (adding a *jsessionid* parameter), you can use the *response.encodeURL(String)*, with the URL-string to encode as parameter. When you call *encodeURL* the server will not change the URL if the cookie mechanism is used to exchange the session id with the browser.

Note the following disadvantages:

- every page sent to the client has to be dynamically generated
- bookmarked links can contain session id’s that don't exist anymore

This optional reading shows you how to disable url rewriting for robots, in order to improve your SEO. It uses a *filter* and you may want to first read the filter chapter.

3.3.3.3. SSL Session information

When using the secured version of the HTTP protocol HTTPS the underlying SSL-protocol creates a secured connection between the client (browser) and the server. In this case the webserver will use the session information form SSL to track its own sessions. No additional mechanisms are required anymore.

3.3.4. Destroying Sessions

Session objects take resources on the server. It depends on the amount of different users you have, and on the amount of information you put in each session.

We don’t want sessions to stick around longer that necessary. But, the HTTP protocol doesn’t have any mechanism for the server to know that the client is gone (meaning: has left the site, his browser crashed, walked away from its pc, …). But the container should be able to know when it’s safe to destroy a session.

3.3.4.1. Setting Session Timeout

Luckily, you do not have to track all the session activity yourself: the container destroys the inactive sessions for you. However, you can configure how long you want to keep a session that’s inactive.

This can be done in two ways:

- Programmative (for a specific session): HttpSession.setMaxInactiveInterval(). Use it when you want to give some users (as administrators) longer sessions. When using this method you set the value of session timeout in seconds.
- Declarative (in the web.xml, for all sessions). In deployment descriptor, you define session timeout in minutes. Consider the example below.

Source

```
<web-app>
   <servlet>
   ....
   </servlet>
   <session-config>
       <session-timeout>30</session-timeout>
   </session-config>
</web-app>
```

Please note that you can set session timeout only using the web.xml descriptor (or *HttpSession* interface). There's no way to change timeout value using merely the annotations.

3.3.4.2. Invalidating the Session

Another way to let the container know it’s safe to destroy the session, is by invalidating it through the method *HttpSession.invalidate()*.

3.3.5. Activity: Store a Shopping Cart

3.3.5.1. Features

The output of BookDetail servlet should be transformed into a form with an "Add to Cart" submit button.

For the purpose of this exercise, the "ShoppingCart" simply consists of a collection of Book objects (no customer identification, neither quantity).

- A book may be added to the cart multiple times.
- A cart should be created if it doesn't exist in the session yet.
- Use URL encoding in the BookCatalog servlet.

3.3.5.2. Steps

We'll first make the a link "add to cart", because it is simpler. When it works, we'll turn it into a button.

1. Write a blackbelt.domain.ShoppingCart class with:
    - a single attribute: *List contents*
    - a public method void add(Book aBook)
    - a public method List getContents()
2. BookDetailServlet should add a hyperlink to the response:
    - link goes to the AddToCartServlet (created below)
    - contains the text "Add book to cart"
    - has a parameter named "isbnToAdd" with the isbn of the book
    - Also add a hyperlink "Back to Catalog"
3. Write a AddToCartServlet to process the link you just added:
    - The concerned book's isbn is sent as the "isbnToAdd" request parameters.
    - Get the corresponding Book object from the BookList.
    - Get the HttpSession instance from the request.
    - Get the ShoppingCart from the HttpSession.
    - If a ShoppingCart cannot be found in the Session, create one and add it as a session attribute.
    - Add the book to the cart.
    - After processing, display a confirmation message like "Book " + book.getTitle() + " added to the cart!"
    - Add a hyperlink "Back to Book Detail"
4. Add AddToCartServlet's definition and mapping to web.xml. You may like to map your servlet to the "AddToCart" url.

Test your code. Then let's change the link into a button You shouldn't need to change the AddToCartServlet. All the changes are made in BookDetailServlet. Change your link "Add book to cart" into a form with:

- "POST" as method
- "AddToCart" as action (the url mapping of AddToCartServlet)
- a "Add to Cart" submit button
- a hidden field named "isbnToAdd" to store the isbn value (that's a trick not covered in the theory), look for "html hidden field".
- Add a hyperlink "Back to Catalog"

Optional additional activity 1: Test your code with cookies disabled in your browser and adjust your code to get it working again, if it doesn't work straight away.

Optional additional activity 2: Add a link, for example in the output of the BookCatalogServlet, to clean-up the session. Use the invalidate method.

3.3.5.3. Hints

- To write the AddToCart servlet as described, you'll have to use:
  - HttpServletRequest.getSession()
  - HttpSession.getAttribute(String) / setAttribute(String, Object)
  - HttpServletRequest.getParameter(String)
- For the optional exercise: Think about URL Rewriting.

3.4. Cookies

3.4.1. Definition

A Cookie is a text information created by a web server. It is stored on the hard drive of the client by the web browser. It is systematically sent within the headers of each HTTP request from that client to that server. Therefore it should be kept short to not degrade performance.

A very typical usage of cookies is to store the preferred language of the visitor to a multi-lingual web site. When the user reconnects to the home page of such a web site a few days later, the cookie with the language identifier is still sent in the header of the HTTP request and the server can directly show the page in the right language.

Another typical usage is to store the credential of a user (username and password) to enable a returning user to auto-login.

3.4.2. Creating a Cookie

The javax.servlet.http.Cookie class enables your server-side code to create and change cookies.

The following code creates a cookie named "language" with the value "FR". It adds it to the HttpResponse object. It will be include in the HTTP response returned to the client which will store it on it's hard drive.

```
Cookie langCookie = new Cookie("language", "FR");
response.addCookie(langCookie);
```

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: language=FR

(content of page)
```

3.4.3. Retreiving Cookies

Cookies are sent with every HTTP request from that browser to the server that requested the cookie creation:

```
GET /MyServlet HTTP/1.1
Host: www.example.com
Cookie: language=FR
Accept: */*
```

The HttpServletRequest.getCookies() returns an array with all the cookies sent in the header of the HTTP request. You need to interate over that array to retreive a specific cookie, because there is no method to retreive them by name.

The code below retreives in the variable *langCookie*, the cookie with the name "language".

```
Cookie langCookie = null;
Cookie[] cookies = request.getCookies();
if (cookies != null) {
   for (Cookie cookie : cookies) {
       if ("language".equals(cookie.getName())) {
           langCookie = cookie;
           break;
       }
   }
}
```

3.4.4. Cookie Attributes

The Cookie class has the following properties:

- **Name**: The identifier of the cookie. Each cookie for a domain must have a unique name. In our example the name is "language".
- **Value**: The String value of the cookie. In or example it is "FR".

- **MaxAge**: A cookie is not meant to last forever. You specify the amount of seconds for your cookie to remain on the hard drive of each client. It is typically a duration of several days, to enable your server to retreive a specific value for a returning visitor. It can also be 0, to mean that the cookie disappear as soon as the visitor closes its browser.
- **Domain**: The domain name to which the client must send the cookie. It enables the client to be selective and to partition the cookies and do not send informations set by one web site to another web site. For example, you would not want the information that your application stores on the browser, to be sent to other sites that the user visit. This domain name is automatically set by the web container.
- **Path**: If your web domain includes many applications, you may not want that each application's cookies to be sent for other applications as well. Because cookies are sent with every request, it is important for performance to limit their amount. With the path, you can set a cookie to be sent to specific URLs only within your domain.
- **Comment**: An optionnal comment associated with the cookie. This is rarely set.
- **Secure**: As cookies goes with every HTTP requests, their value can be seen by hackers spying the network. It's probably not a problem for preference cookies such as the language. You may want to store more sensitive information in cookies. In that case you may require that the browser only send these cookies over an encrypted connection. When the *secure* property is true, that cookie is only send over an HTTPS connection.

3.4.5. Disabling Cookies

Web browsers have the option to disable cookies. Be aware that some of your visitors do not store the cookies to send. How to disable cookies depends on the browser.

For IE 9, click on the tools button at the right on the tool bar, and select in *Internet Options...* in the menu. Go to the privacy tab and select the Advanced button.

For Chrome, enter that url: chrome://settings/content

For FireFox, XXXXXXXXXXXX TO BE COMPLETED.

SCREENSHOT IMAGE XXXXXXXXXXXXXXX

3.4.6. Exploring Cookies

Your web browser can list all the cookies it has stored. The procedure is different for each browser.

IE (version 9 included) is not very practical on that point. It will list you files, one file per cookie, on your hard drive.

Click on the *Tools* icon in the main menu area. From the menun select *Internet Options...*. In the dialog box, go to the *General* tab. In the *Browsing history* section, press the *Settings* button. In the new dialog box, press the *View Files* button to open the file explorer. Scroll down until you see the files labeled as cookies.

Chrome is much easier. Just go to that URL: *chrome://settings/cookies*. Domain names storing cookies on your hard drive will be listed. You can double click on any of them to list its cookies, and click on any cookie to see its content.

XXXXXXXXXX Procedure for FireFox.

3.4.7. Activity: Store the User Name

In this activity, you enable the user to fill his name in a form, then you remember the name in a cookie. In the BookCatalogServlet, you retreive and display that name.

Start from your current book application.

1. Create the NameFormServlet to show a form with one field (the user name) and one submit button.
2. Create the StoreNameServlet to process that form. It created a cookie with the name "username", value from the form field and age of 60 seconds. The submit button of the form sends the request to the StoreNameServlet.
3. In the BookCatalogServlet, verify the presence of the username cookie. If it is in the request, add a text on the page "Wecome back, XXX" where XXX is the name of the user.
4. Restart your web server to test your form. Go the the StoreNameServlet, fill the form, press the submit button. Then (within the next 60 seconds) go to the BookCatalogServlet and see your name displayed.
5. Wait to enable the 60 seconds delay to expire (age of the cookie) and refresh the BookCatalogServlet page. Your name should not be displayed anymore.
6. List your browser's cookies for your domain *localhost*. The cookie should be disappeared, and your localhost domain is probably not even listed.
7. Fill the form of the NameFormServlet again and press the submit button to recreate the cookie.
8. Check your browser's cookies again for localhost. You should find it now (within 60 seconds).

XXXXXXXXXXXXXX SCREENSHOT of IE9 DISPLAYING THE COOKIE.

XXXXXXXXXXXXXX SCREENSHOT of Chrome DISPLAYING THE COOKIE.

Optional part: Inspect the HTTP header when you press the form submit button. Check the HTTP response that contains the new cookie. Check the next HTTP request sent to the server, it should contain the cookie as well. To view HTTP headers, press F12 in IE9 or Chrome, as you learned in the HTTP topic.

3.5. ServletContext

In the previous topics, you learned to use the request and session scopes.

In this topic, you will use the third one, namely the ServletContext scope. You will enhance the internal architecture of your book application to store the global list of books in that scope.

You will also use a listener object to initialize and attach the list of books to the ServletContext, when the application starts up.

3.5.1. Global Scope

The ServletContext is a global scope that is accessible for all the parts of your web app, so not only during a specific client-session (session scope) or during a specific request of one client (request scope). It's also named the *web application* scope. You will have one ServletContext instance for your whole application.

It is a good place to store any global information about your application, as for example, the list of books in our catalog for our book application.

Note that it is slightly different from static attributes (JVM scope) because you could have multiple web applications inside one JVM (depending on how you configure Tomcat). On the contrary one clustered web application accross mutliple JVMs (accross multiple computers), has one ServletContext per JVM. If you need an even more global scope for clustered environement, you need to rely on another mechanism not provided by the web container, as an external database.

3.5.2. ServletContext Interface

The servlet context scope is represented by the javax.servlet.ServletContext interface. The the web container provides an implementation class and single instance of it.

The ServletContext interface, includes the following methods:

setAttribute()
getAttribute()
getAttributeNames()
removeAttribute()
	Manage the attributes that you attach to the servlet context scope.

getInitParameter()
getInitParameterNames()
	Get parameters from the deployement descriptor.

getResource()
getResourceAsStream()
getResourcePaths()
	Ways to read files that are stored in the web application, such as those inside the war packaged file.

3.5.3. Retrieving the ServletContext

You can obtain that instance from the ServletConfig object or from any Servlet (through its GenericServlet ancestor).

The following code will display "sc1 == sc2 : true", because the same ServletContext is returned by both methods.

Source

```
@WebServlet("/HelloWorldServlet")
public class HelloWorldServlet extends HttpServlet {

   @Override
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
       PrintWriter out = response.getWriter();
       ServletContext sc1 = this.getServletConfig().getServletContext();
       ServletContext sc2 = this.getServletContext();
       out.println("sc1 == sc2 : " + sc1 == sc2);
   }

}
```

3.5.4. Managing Attributes

As you did with the HttpSession object, you can attach attributes to the ServletContext. These will be available for the other parts of the application.

```
ServletContext sc = this.getServletContext();
sc.setAttribute("currentGlobalRebate", new Integer(10)); // 10% discount from now.
```

```
ServletContext sc = this.getServletContext();
Integer discount = sc.getAttribute("currentGlobalRebate");
```

This way, you can use the ServletContext object to store values global to your application. You will need to do this much less often than attaching attribute to the HttpSession, but you will need it soon or later in a medium to large web application.

You may also use the removeAttribute() method to remove an attribute.

3.5.5. ServletContext Init Parameters

It may be convenient to store application parameters outside the Java code. For example these parameter's value could be changed according of the execution environement (development, test, production).

You can initialize parameters, accessible for all the parts of your web app, in the web.xml.

```
<web-app ...>
   ....
   <context-param>
       <param-name>supportEmail</param-name>
       <param-value>support@MyCompany.com</param-value>
   </context-param>
</web-app>
```

They can be accessed via the ServletContext object’s *getInitParameter()* method.

```
String value = this.getServletContext().getInitParameter("supportEmail");
```

3.5.6. Servlet Init Parameters

You can also set init parameters for a specific servlet. It if ofen the case when you use third party servlets that need to be configured. In the example below, we configure the ApplicationServlet from the Vaadin framework. We specify the name of the class that resides in a jar file that we've put in the web application. We also specify one init parameter which name is "application" and value is "MyApplicationClass". That third party servlet expects you to provide a value for the "application" parameter, else it cannot work an will throw an error message.

Source

```
<web-app ...>
   ....
   <servlet>
       <servlet-name>VaadinServlet</servlet-name>
       <servlet-class>
           com.vaadin.terminal.gwt.server.ApplicationServlet
       </servlet-class>
       <init-param>
           <param-name>application</param-name>
           <param-value>MyApplicationClass</param-value>
       </init-param>
   </servlet>
</web-app>
```

Init parameters are retreived through your *HttpServlet.getServlet**Config**()* method. In the ApplicationServlet, you would write that line of code:

Source

```
String value = this.getServletConfig().getInitParameter("application");
```

3.5.7. Servlet Init Parameters via Annotations

It is also possible to define these parameters inside the Java code through annotations. But you probably never want to do that because the advantage of parameters (over classic Java constants in the code) is to externalize the values from the code to ease their change.

Source

```
@WebServlet(name="VaadinServlet", urlPatterns={"/*"},
       initParams={ @WebInitParam(name="application", value="MyApplicationClass")
public class ApplicationServlet extends javax.servlet.http.HttpServlet {
       ...
}
```

3.5.8. Initializing a Web Application

As your web application grows, you will need soon or later to run some initialization code when the web container starts.

For example, imagine we want to initialize a DataSource for connecting to a database (or want to initialize Spring and/or Hibernate). This must be globally accessible in the web application, so we will use a ServletContext attribute for storing it.

You could store the DataSource lookup name as a context init parameter, but which servlet will be responsible for looking up de DataSource and storing it as an attribute in the context? Do we really want to try to guarantee that one servlet will always run first?

The ServletContextListener interface will enable you to define that initialization code.

3.5.9. ServletContextListener

ServletContextListener is a call-back interface that you register on the web container. When the web container starts your web application, it calls the contextInitialized() method of your implementation of ServletContextListner.

In fact your object will be notified:

- when the context is initialized (the application is being deployed) and
- when the context is destroyed (the web app is undeployed or goes down)

The container (Tomcat) knows which method of this listener needs to be called in each situation.

Source

```
@WebListener
public class MyServletContextListener implements ServletContextListener {
   public void contextInitialized(ServletContextEvent event) {
       ServletContext sc = event.getServletContext() ;

       //... do you initialization stuff here ...
   }
   public void contextDestroyed(ServletContextEvent event) {
       // nothing to do here
   }
}
```

3.5.10. ServletContextListener in the Deployment Descriptor

Alternatively, if you don't annotate your class with @WebListener, you need to define it in the deployement descriptor using a element.

```
<web-app ...>
   ....
   <listener>
       <listener-class>
           com.example.myapp.MyServletContextListener
       </listener-class>
   </listener>
</web-app>
```

NOTE: There are other types of listeners. E.g. the HttpSessionBindingListener can be used to let you know when an attribute has been added or removed from a session.

3.5.11. Accessing Resources

You will sometimes need to read/write files within your web application. A typical case is to read a configuration text file.

You could just use the classic java.io API as for any Java (SE) application. But you face two problems:

- You (as programmer) are not supposed to know upfront where the deployer of the application will install the web server on the file system, and you don't know the absolute path to your files. It would be a bad practice (and violate the servlet specification) that your web application depends on a concrete installation and configuration of the web container.
- The web container does not necesarily unzip the war file on the file system when deploying the application. If a configuration file is in the zipped war file, it is not accessible that easily as a normal file on the file system would be.

The ServletContext interface provides method to help you accessing web application files.

3.5.12. Reading Files

You read files from your web application using the class loader, because it is able to access the content of the (maybe not unpackked) war file. The cloass loader is normally responsible to read the .class compiled java file. But it can also read any file, as for example, a text configuration file.

The following example uses the ServletContext.getResourceAsStream() method to read a text .properties file. It will (internally) use the classloader. Such code could be in a servlet or more typically, in a ServletContextListener.

```
InputStream in = servletContext.getResourceAsStream("/WEB-INF/myApp.properties");
Properties properties = new Properties();
ps.load(in);
String connectionString = properties.getProperty("jdbcConnect");
```

The code above gets an InputStream for the file "/WEB-INF/myApp.properties". The path is relative to the root of your web application, which in Eclipse is "YourProject/WebContent/" by default. Once you get the InputStream, the rest of the code is pure Java SE (non web related). In our example, we read the text file below and at the end, connectionString contains "jdbc:mysql://localhost:3306".

Source

```
jdbcUser = root
jdbcPassword = secret
jdbcConnect = jdbc:mysql://localhost:3306
```

3.5.13. Writing Files

Web applications try to limit writing files. Business data is usually written into a relational DB. When you really need to write a file, you first need to choose where to put it: inside or outside the web application.

3.5.13.1. Outside the web application

To write a file outside the web application you just use the Java SE I/O API as for any Java application. You need to know in which folder to write your file, and it's probably a different folder in development or in production. The path of that folder could be a parameter of your application (as a .propoerties file that you read, or an init parameter).

If you need to write a temporary file, that should be cleaned when the web application stops, you can use the system property "java.io.tmpdir" which will be setup by your web container with a correct value.

Source

```
String destinationDirectory = System.getProperty("java.io.tmpdir");
```

3.5.13.2. Inside the web application

Technically, when the war file is kept unzipped by your application server it is not possible to write a file in your web application directory. Even if you unzip your war file, everytime you redeploy your application these directories will be replaced and the files written by your application would be lost.

But sometimes, you need to write files that must be served. For example, you want to display in the web browser of a user, its picture that he just uploaded. Or you want to generate a pdf file and make it accessible to your application users.

A way to acheive that is to write the file in a directory outside your web application, and to use an apache server in front of your web container. Let's suppose that you write the file userimage123.jpg in the c:\gen-content directory, outside your web application folder. Of course you backup that directory as you would do for any application data that you don't want to lose. You configure a rule in your apache server that takes all request for "\gen" path and make them take files out of the c:\gen-content directory. It would be the case of the request to "http://localhost/gen". All other requests are forwarded to your web container.

In the step A, the web container creates the file for some reason (probably a user just uploaded its picture).

Much later, in the step B1, a user requires to access that picture. Apache does not forward that request to the web container because the URL path starts with "/gen". As step B2, Apache web server sends the file as response to the client.

3.5.14. Activity: Initialize BookList Correctly.

Up to now, we stored the *BookList* instance in a static attribute of the *UglyGlobalVar* class (defined in the *Create the BBStore Project* topic). We'll change the situation and store it in the *ServletContext*. People could argue that it worked well so far with the static attribute. But it is considered as a bad practice for technical reasons that were historically correct and that are now debatable.

The BookList instance should be created when the BBStore application starts up and be registered as a context attribute, in order to make the servlets able to find it at one common place.

Steps:

1. Create a new com.example.bbstore.web package.
2. In that package, create a class named BBStoreListener implementing ServletContextListener and annotated with @WebListener.
3. In its contextInitialized() method, construct the BookList instance and put it as a ServletContext's attribute. The ServletContext instance may be retrieved from the ServletContextEvent parameter.
4. In the same package, create a new *WebUtil* class
5. Implement this method in the WebUtil class. It gets the BookList instance from the ServletContext (passed as parameter).
   ```
   public static BookList getBookList(ServletContext context)
   ```
6. Adapt BookDetailServlet: retrieve the BookList from the context when needed, by using your new *WebUtil.getBookList()* method.
7. Delete the class UlgyGlobalVar that should not be used anymore.
8. Test that your application works as well as before.

3.6. Selecting the Right Scope

3.6.1. Overview

The servlet specification identifies 3 scopes:

- Request
- Session
- ServletContext = Application

The reality is more subtle. The table below summarizes the possible scopes in a web application:

During a request	request, ThreadLocal
Accross 2 requests for one page	link parameter, hidden field

Accross requests for one user	session, cookie
For all users	ServletContext

It is very difficult to put these scopes into a static diagram, because the important comparaison is the relative lifetime of each element.

3.6.2. Request

The shortest scope is the request. The servlet specification offers the (Http)ServletRequest as a place to store attributes, each with a name and a value. This is fine for values needed by servlets because the servlets get a reference to the ServletRequest object.

However, during the lifetime of a request, many other classes can be called. Your servlet is a starting point, and can call business logic methods, which can also call other business logic methods. For example, a servlet may be responsible of displaying a promotion to the user. What promotion depends on the buying habits of that user and is computed by a complex business logic algorithm. That algorithm will not reside in your servlet class, but into some specialized business logic oriented classes. What if that algorithm needs data from the request scope? It would not be clean to pass the HttpServletRequest instance to that business logic. The business logic is not supposed to depend on any UI (servlet) object. In such case, the ThreadLocal pattern is helpful, because each requests corresponds to one thread. You will learn about threads in web applications in the next lesson.

Typical objects placed in the request scope include:

- Objects retrieved from the DB by one servlet, and displayed (HTTP response) by another servlet.
- The DB connection (or JPA EntityManager) used to find/save data during that request.

3.6.3. Accross 2 requests

Sometimes, a servlet wants to pass information to the next servlet, but some user interraction is required before activating the second servlet.

It is typically the case when a servlet puts a parameter into a link. For example, the BookCatalogServlet puts the isbn to identify a book, in the links to the BookDetailServlet. When the user clicks, the BookDetailServlet is activated and can retreive the isbn of the concerned book.

Form buttons have no parameter. In the case of forms, hidden fields are used.

Typical information placed in this scope includes:

- primary key of DB records, to retreive the corresponding record easily.

3.6.4. Accross all Requests for One User

The next scope, in terms of lifetime, keeps the data longer but is narrowed to a specific user (one browser on one computer).

Cookies enable to keep very small piece of data for a long time, typically several days. It's convenient to store information that the application would like to get when the visitor comes back. That information is stored on the browser.

Typical information placed in a cookie includes:

- Session id (for associating the right HttpSession)
- Preference options, such as language and number or search results to display.
- Tracking information, such as knowing if it's the first visit of a user (to tune advertizing).

HttpSession enables to keep larger amount of information but for a shorter time, typically less than one day, until the visitor has finished its visit.

That information is stored on the server and consumes memory. For scalability, it cannot be kept too long.

Typical information placded in the HttpSession includes:

- The user logged in (User object or DB primary key)
- Conversation data holders, such as a shopping cart, or a wizard form spanning multiple pages.

3.6.5. For All Users

The last scope, ServletContext, lasts as long as the web application is up and running, and is shared by all requests (and therefore all users).

Typical information kept at that level includes:

- Global configuration objects and factories, such as the Spring ApplicationContext instance, or the JPA EntityManagerFactory
- Global business data kept in memory to minimize the number of DB access.

3.7. Optional: Spring Integration

This topic is optional, and implies that you know how to use Spring dependency injection.

In a regular Java application you initialize Spring from the main method (or a method called by the main method). You also get your first bean (first of an IoC graph) explicitely from the ApplicationContext.getBean(String) method.

In this topic, we see how these processes typically happen in a web application.

3.7.1. Initialization

3.7.1.1. Doing it Yourself

This is the typical Spring initialization code of a Java SE program:

```
public static void main(String[] args) {
   ApplicationContext myApplicationContext =
    new ClassPathXmlApplicationContext("applicationContext.xml");
   BankService bankService = (BankService)applicationContext.getBean("bankService"
....
}
```

You just insert the ApplicationContext instantiation in your program flow.

For a web application, the suitable place to put this code is a ServletContextListener. It is executed once when your application starts.

You also need a place to put the ApplicationContext once instantiated. It is typically placed as an attribute of the ServletContext.

If you had to write this code yourself, it would probably be similar to this:

```
public class MyServletContextListener implements ServletContextListener {
   public void contextInitialized(ServletContextEvent event) {
       ApplicationContext myApplicationContext =
        new XmlWebApplicationContext("/WEB-INF/applicationContext.xml");

       ServletContext sc = event.getServletContext() ;
       sc.setAttribute("ApplicationContext", myApplicationContext);
   }

....
}
```

3.7.1.2. Using Spring ContextLoaderListener

Fortunately, Spring provides such a ServletContextListener out of the box: *ContextLoaderListener*
You just need to declare it in your deployment descriptor.

```
<listener>
   <listener-class>
       org.springframework.web.context.ContextLoaderListener
   </listener-class>
</listener>
```

3.7.1.3. Where is Your applicationContext.xml?

3.7.1.3.1. Web Applications

In our implementation of the ServletContextListener above, we used *XmlWebApplicationContext* from Spring, instead of the *ClassPathXmlApplicationContext* used in Java SE applications.

The ContextLoaderListener of Spring also uses (internally) the *XmlWebApplicationContext*.

The *XmlWebApplicationContext* loads the applicationContext.xml file using the ServletContext.getResource() method. That method can only accessfiles inside the web application directory (WebContent in Eclipse). By default, Spring looks for *YourProject/WebContent/WEB-INF/applicationContext.xml*.

This can be changed via (servlet)context parameters:

```
<context-param>
 <param-name>contextConfigLocation</param-name>
 <param-value>
  /WEB-INF/classes/applicationContext.xml
 </param-value>
</context-param>
```

In the example above, Spring will look in */WEB-INF/classes/* instead of just */WEB-INF/*.

3.7.1.3.2. Non-web Programs

Beside your web application, you probably have batches to be executed, for example to send daily mails to your users, or to do some daily computation in the DB, or simple to be executed once during a version migration,etc.

For a batch application (a normal Java SE program with a static main method), the most healthy way to get the applicationContext.xml is probably through a ClassPathXmlApplicationContext class looking in the class path for the file. And the most default natural place to put your file at the root of your *src* directory.

The Java compiler will not understand the xml file and just copy it i in its output folder. If the output folder is *YourProject/WebContent/WEB-INF/classes*, the complied code (.class files) and the applicationContext.xml file will be placed there by the Java compiler.

This will work:

```
public static void main(String[] args) {
   ApplicationContext myApplicationContext =
    new ClassPathXmlApplicationContext("applicationContext.xml");
   ....
}
```

In our example, the applicationContext.xml is both:

- in the web app directory (WebContent): XmlWebApplicationContext can load it
- in the classpath: ClassPathXmlApplicationContext can load it

3.7.1.3.3. Eclipse Default Output Folder

By default, with a freshly created Eclipse *Dynamic Web Project*, the output compilation directory is *MyProject/build/classes*. It is outside the *MyProject/WebContent* directory tree, with the consequence that Spring's *ContextLoaderListener* will not be able to load it.

To solve that problem, we will do something very natural: change Eclipse output compilation directory to the correct place (correct according to the Java EE specification as seen in chapter *Web Application*): *MyProject/WebContent/WEB-INF/classes*. Do that change very simply: right click on your project > properties > Java Build Path > source > Default output folder.

Then, your applicationContext.xml file is under the WebContent directory, and Spring's ContectLoaderListener will be able to load it.

3.7.2. Getting Beans

3.7.2.1. From ServletContext

To obtain a Spring managed bean from another Spring bean, it is easy, you just wire them in Spring with IoC. For example, you have an instance to a Dao in a Service class.

But how to get the first Spring managed bean of a chain? How to get your Service bean for example?

You will get it by calling the *ApplicationContext.getBean(String)* method. As we have just seen, your ApplicationContext instance is placed in your ServletContext by the org.springframework.web.content.ContextLoaderListener class. Spring provides a utility method to retrieve the

ApplicationContext from the ServletContext: *WebApplicationContextUtils.getRequiredWebApplicationContext(ServletContext)*.

```
ApplicationContext myApplicationContext =
 WebApplicationContextUtils.getRequiredWebApplicationContext(servletContext);
```

3.7.2.2. Autowiring

Some web frameworks propose Spring integration. It mostly means that they will do the autowiring of the first bean in the chain automatically with the objects they manage.

If Java EE provided Spring integration (it does not) it would be as if your Servlet were created with references to Spring beans initialized. Something like:

```
public class MyServlet extends HttpServlet {
  @Autowired
  BankService bankService;

  public void doGet(HttpServletRequest request, HttpServletResponse response) {
      .... just use bankService, it has been assigned ;-) ...
  }
}
```

Unfortunately, it does not work with Servlet. But it's no big trouble as nobody directly develops with Servlets. Struts v2, Tapestry, Wicket, JSF, etc. provide Spring integration. For example, Struts 2 actions can be wired to Spring managed beans.

If you really need Spring injection in your servlet, you might consider using @Configurable with AOP weaving. But the setup is probably an overkill.

3.7.3. Exercise

Turn your web application into a Spring enabled web application by following the steps of this chapter.

You will need to:

1. Download the jars of Spring core and its required dependencies.
2. Put them in your WEB-INF/lib directory; and Eclipse will automatically add them to your build path.
3. Create a *Main* class with a *main* method to have a regular (non web) program for testing.
4. Create a Spring applicationContext.xml file
5. From your Main.main() method, get the ApplicationContext and from it, get a bean, to verify that Spring works well with Java SE (outside the web).
6. Declare the ServletContextListener provided by Srping in the web.xml (it will instantiate your ApplicationContext and put it in the ServletContext).
7. In one of your servlets, get the ApplicationContext from the ServletContext, and from that ApplicationContext get a bean of Spring.

8. Test your modified servlet.

4. Processing Requests

You have seen how to store data in various scopes. For example, we are able to build a login facility and store business objects at the application level.

Sometimes, you need to put more complexity in the processing of your request. For example, when a user successfully logs in, you want to forward the request to another page; but when the login fails, you want to display the login form again. Another example is to perform some security check systematically at every request, in order to redirect the non logged in users to the login page.

In this lesson, we create filters, we forward and redirect requests and include reusable fragments in pages. We also learn how our web application is multi-threaded and why we need to care. Finally, we'll make a small Ajax application where requests are smaller and require a different processing.

4.1. Filters

Sometimes, you want to perform operations on all request/responses, systematically.

For example, you would like to add some HTTP headers to the responses of all your servlets. In this specific example, we change headers to tell firewalls and browser to not cache the pages of our application. More information about caching and HTTP Header can be found in the HTTP 1.1 Specification or a Tutorial on Caching for Webmasters

We will place a filter that will be executed before any servlet for every new request.

4.1.1. Java Code

Our class, a filter, implements the *javax.servlet.Filter* interface. We don't need the *init* and *destroy* methods in this example and we leave them empty.

Source

```
public class ResponseHeaderFilter implements javax.servlet.Filter {

   @Override
   public void doFilter(ServletRequest servletRequest,
            ServletResponse servletResponse, FilterChain filterChain)
            throws IOException, ServletException {

        HttpServletResponse response = (HttpServletResponse) servletResponse;
        //Set the date of this response
        response.setDateHeader("Date", new Date().getTime());
        //Set all the pages to expire at a older date (they are already expired)
        response.setDateHeader("Expires", 0);
        // Set standard HTTP/1.1 no-cache headers
        response.setHeader("Cache-Control","no-cache, must-revalidate, s-maxage=0,
        // Set standard HTTP/1.0 no-cache header
        response.setHeader("Pragma", "no-cache");

        filterChain.doFilter(servletRequest, servletResponse);
    }

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
    }

    @Override
    public void destroy() {
    }

}
```

The interesting method is *doFilter*. We get the request and response as parameter, and also a *FilterChain* instance. We are responsible to start the processing of the request handling by calling FilterChain.doFilter(...) ourselves. That way, the filter can do something before the servlet is called, and something after the request is called. In our example above, we only do something before (we set headers on the response) and nothing after.

After modyfing the response with the filter, the former may look like:

Source

```
HTTP/1.1 200 OK
Date: Mon, 12 Mar 2011 19:00:00 GMT
Server: Apache/1.3.12 (Unix) Debian/GNU mod_perl/1.24
Content-Length: 3369
Expires: 0
Cache-Control: no-cache, must-revalidate, s-maxage=0, proxy-revalidate, private
Pragma: no-cache
Connection: close
Content-Type: text/html

Response content goes here.
```

Please note that headers specified on the filter level, has been added to the response.

4.1.2. Chaining

Filters can be chained. You can apply multiple filters to all the requests. Simply declare them in the web.xml file and they will be called in the order of the position of their mapping declaration.

The diagram below demonstrates the flow of information in the filters chain.

4.1.3. Filter annotations

Starting from the version 3.0 of the Servlet specification you can replace XML definition of the Servlet Filters with the annotations.

The example below demonstrates how to configure filter class with deployment metadata.

```
@ServletFilter
@FilterMapping("/foo")
public class FooFilter {
 public void doFilter(HttpServletRequest req, HttpServletResponse res) {
       .....
 }
}
```

4.1.4. Deployment Descriptor

Non annotated filters are declared in the web.xml file.

```
<web-app ...>
   ....
   <filter>
       <filter-name>ResponseHeaderFilter</filter-name>
       <filter-class>com.example.web.ResponseHeaderFilter</filter-class>
   </filter>
</web-app>
```

You must also tell some url mapping. Here we enable the filter for all the requests.

```
<web-app ...>
   ....
   <filter-mapping>
       <filter-name>ResponseHeaderFilter</filter-name>
       <url-pattern>/*</url-pattern>
   </filter-mapping>
</web-app>
```

But we could say, that we only want the filter for requests to the html files in the *foo* directory.

```
<web-app ...>
   ....
   <filter-mapping>
       <filter-name>ResponseHeaderFilter</filter-name>
       <url-pattern>/foo/*.html</url-pattern>
```

```
    </filter-mapping>
</web-app>
```

4.1.5. Exercise

Write and test a filter that counts the requests. Display the current counter at the console every time a request hits your server.

Also think about answer the following questions. Where would you store hits count for the given Servlet? Where would you store hits count for whole application? To particular user?

4.2. Forward & Redirect

4.2.1. Need

A common practice in most web application (using Servlets, JSF, Struts or something else), is to transfer the request to another "page". In our case, we can transfer a request from a servlet to another servlet.

For example, the visitor clicks the "Add to cart" button in our BBStore application. The AddToCartServlet may decide that if the visitor is not logged in, we should show the login form. We do that by forwarding or redirecting the request from the AddToCartServlet to the LoginServlet.

Source

```
public class AddToCartServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        User user = (User) request.getSession().getAttribute("user");
        if (user == null) {
            ..... forward to login servlet
        } else {
            ..... add to cart
        }
    }
}
```

4.2.2. Forward

The forward mechanism is internal to the application server (Tomcat). It is like an internal call from a servlet to an internal resource. You could forward, for example, to an html file of your application. It is also common to forward to a JSP (out of the scope of this course).

To forward, you need an instance of RequestDispatcher, available through the HttpServletRequest object.

```
request.getRequestDispatcher("Servlet2").forward(request, response);
```

To get the RequestDispatcher, you need to specify the target url. In our example, it is an url from the root of your web application. If the full path to Servlet2 is

```
http://localhost:8080/MyWebApp/Servlet2
```

then the root of the web application is

```
http://localhost:8080/MyWebApp/
```

In the following example, Servlet1 forwards to Servlet2:

```
public class Servlet1 extends HttpServlet {
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
 request.getRequestDispatcher("Servlet2").forward(request, response);
}
}
```

```
public class Servlet2 extends HttpServlet {
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
               throws ServletException, IOException {
       PrintWriter out = response.getWriter();
       out.println("Hello World");
}
}
```

You enter the URL to Servlet1 in your browser and get:

- The result: "Hello World"
- With the url: http://localhost:8080/MyWebApp/Servlet**1**

Because forward is an internal server-side (hidden) mechanism, the browser does not know that Servlet2 (and not Servlet1) sent the response back. The browser url bar still shows "Servlet1".

4.2.3. Redirect

On the contrary, with a redirect (instead of a forward), the browser's url bar would show "Servlet2".

Source

```
public class Servlet1 extends HttpServlet {
protected void doGet(HttpServletRequest request, HttpServletResponse response) thr
 response.sendRedirect("Servlet2");
}
}
```

By calling *sendRedirect()* on the HttpServletResponse object, you send a response to the browser, with no content and a special code in the http header. The header also contains the url of Servlet2, and the browser is requested to immediately send a request for Servlet2 (with no user action).

You enter the URL to Servlet1 in your browser and get:

- The result: "Hello World"
- With the url: http://localhost:8080/MyWebApp/Servlet**2**

4.2.4. Exercise

Write and test two servlets:

- one that forwards requests to a JSP page
- one that redirects requests to a JSP page

Write a simple JSP page and add the following:

- a hyperlink to the forwarding servlet
- a hyperlink to the redirecting servlet
- the referring URI
- the referring page's HTTP result code (trickier than you think!)

Optional:

- create a generic request handler servlet and make the two servlets into subclasses
- create a counter to track how many times each servlet has been invoked
- add as many fields as you can think of that may give different results between forward and redirect
- how about adding a timer to measure the difference in speed per request?

4.3. Include

4.4. Threads

4.4.1. One Thread per Request

A Fundamental concept in web applications is that the web container associates one thread to every request.

It means two important things for the programmer:

- Your code must be threadsafe
- You can associate request scoped values to the thread

4.4.2. ThreadSafety

Multiple requests may (and will very probably) hit your server simultaneously. They may even get into the same servlet. The code in your servlet must be threadsafe. It means, for example, that you probably defined no attribute in your servlet class, and prefer local variable and parameters.

If you do web programming, you cannot simply ignore threadsafety. Defining threadsafety is out of the scope of this course.

The request and response instances have no problem, because your thread is alone to work on them. On the contrary the HttpSession and ServletContext objects are shared by multiple threads.

There is two levels of threadsafety here:

1. Are these Tomcat objects threadsafe? Some books tell that the HttpSession object is not threadsafe (means that their internal structure could be broken by multi threaded calls). Servlet specification 2.5 clarifies that and now they must be threadsafe. Recent Tomcat 5 and 6 versions are threadsafe on this point.
2. Is your application threadsafe? As for any shared instance in multithreading, your usage of HttpSession can or cannot be threadsafe.

For example, this code is thread safe (because of 1):

Source

```
session.setAttribute("key", value);
```

This code is not threadsafe (because of 2):

```
    cart = (ShoppingCart) session.getAttribute("cart");
    if (cart == null) {
        cart = new ShoppingCart();
        session.setAttribute("cart", cart);
    }
```

This code is threadsafe:

```
HttpSession session = request.getSession();
ShoppingCart cart;
synchronized (session) {
    cart = (ShoppingCart) session.getAttribute("cart");
    if (cart == null) {
        cart = new ShoppingCart();
        session.setAttribute("cart", cart);
    }
}
```

To quote a jGuru answer:

That said, I wouldn't think that you would have to worry about this issue much. Since HttpSession objects are unique to each user, you probably won't be accessing any of these with more than one thread at the same time. As for ServletContext objects - nearly all the methods are read methods. And since the only objects you will be putting in the ServletContext are application wide in scope, you will probably want to put these in at the apps startup anyway and read them from that point on.

That said, note that with Ajax, its is common to have your JavaScript code fire multiple request to the server for the same page, wich leads to very probable concurrency.

This IBM article explains the web applications thread safety in details.

Typically your servlet calls the service layer which calls the persistency layer, as described in our architecture overview common chapter. Your service and persistency layers must also be threadsafe.

4.4.3. ThreadLocal

Sometimes, you want to retrieve values that are bound to your request or user issuing the request. And you want to retrieve them from anywhere in your code, not only from the UI/Servlet layer.

For example, to implement an additional security check in the business logic, you may want to know the logged in user. But we have seen in the HttpSession chapter that we store the logged in user in some web related objects (HttpSession) which should be unknown from the business logic. You have two options:

- Pass more parameters (as the logged in user) from the UI to the service layer.
- Store these values (as the logged in user) in a ThreadLocal variable.

ThreadLocal is a Java SE class to store one instance of something (as a User) per thread. See this excellent StackOverflow answer for a web example with a User.

This *ThreadLocal-request* pattern is very common in modern web applications. For example, it is internally used by Spring to store one instance of your JPA EntityManager per thread (yes, you won't mix DB requests from various HttpRequests in the same transaction!).

4.5. Ajax

5. More Details - Servlet

5.1. Servlet API Overview

You have seen many interfaces and classes of the Servlet API, spread in the previous lessons. It is time to get an overview of them and how the main interfaces interact with each other.

5.1.1. Class Diagram

The Servlet API is a set of Java classes and interfaces that define a standard interface between the web container and your web server-side code running inside. It is composed of four packages:

- javax.servlet
- javax.servlet.http
- javax.servlet.annotation
- javax.servlet.descriptor

5.1.2. The Servlet interface

Objects of this class receive HTTP requests and produce HTTP responses.

The javax.servlet.Servlet interface defines (among others) the following methods:

- **init()**: The servlet is initialized only once by calling its init() method, on loading the servlet. The servlet can be loaded after the servlet container process starts (pre-loaded servlet), or when the servlet is first used.
- **destroy()**: Called when the servlet is unloaded.
- **ServletConfig getServletConfig()**: ServletConfig provides access to deployment descriptor parameters of the Servlet. This method returns the ServletConfig object that initialised this servlet.

5.1.3. The HttpServlet class

Servlet is implemented by javax.servlet.GenericServlet which itself is extended by javax.servlet.http.HttpServlet defining these additional methods:

- **doGet (HttpServletRequest, HttpServletResponse)**: Invoked whenever an HTTP GET method is issued for a URL that points to the servlet. Also invoked when a HTML form is posted using a method of GET. A HTTP GET method is the default when a URL is specified in a Web browser, and is the default when submitting a form. The default implementation of this method yields an error.
- **doPost(HttpServletRequest, HttpServletResponse)**: Invoked whenever an HTTP POST request is issued through an HTML form. The default implementation of this method yields an error.
- **service (HttpServletRequest, HttpServletResponse)**: The architecture of the servlet API is that of classic service provider with a service() method through which all client requests will be sent by the servlet container software. The default implementation of the service() method calls doGet() or doPost() depending on the method of the HTTP request. Other methods are also available, like doPut(), but are usually not overridden. (Consequently, invoking doPut() via a HTTP PUT request yields an error.)

5.1.4. The ServletConfig interface

Objects implementing this interface encapsulate configuration data for an individual servlet. Every servlet implements ServletConfig. Surprisingly, servlets have both an inheritance and association relationship with ServletConfig.

- **String getServletName()**: Returns the name of this servlet instance.
- **String getInitParameter(String name)**: Returns a String containing the value of the named initialisation parameter, or null if the parameter does not exist.
- **Enumeration getInitParameterNames()**: Returns the names of the servlet's initialisation parameters as an Enumeration of String objects, or an empty Enumeration if the servlet has no initialisation parameters. These init parameters are usually defined in the web.xml deployement descriptor.
- **ServletContext getServletContext():** Returns a reference to the ServletContext in which the servlet is executing.

5.1.5. The ServletContext interface

Objects implementing this interface encapsulate configuration data for a set of related servlets. Each web application has one servlet context instance. It defines methods that a servlet uses to communicate with its servlet container, for example, to get the MIME type of a file, dispatch requests, or write to a log file.

You can access the ServletContext by calling Servlet.getServletConfig().getServletContext(). The methods include:

- **get/setAttribute, getAttributeNames**: to manage the application scoped variables.
- **getInitParameter, getInitParameterNames**: to read the parameters of the web.xml
- **RequestDispatcher getRequestDispatcher(String path)**: Returns a RequestDispatcher object that acts as a wrapper for the resource located at the given path. Usually invoked before RequestDispatcher.forward().
- **void log(String msg)**: Writes the specified message to a servlet log file, usually an event log.
- **InputStream getResourceAsStream(String)**: provide a read-only access to a file resource located in the web application directory.

The ServletContextListener interface enables you to get notified when the web application is started and destroyed.

5.1.6. The HttpServletRequest interface

The HttpServletRequest interface encapsulates information about the client request, including any data that may have been sent from the client. You can get: request header, content type, length, HTTP method (GET, POST,...).

It implements ServletRequest.

A main role of the HttpServletRequest is to provide parameters passed in the HTTP request, through the methods getParameter(String) and getParameterNames().

Other methods include:

- **getParameterValues(String)**: returns all values for a multi-value parameter, in the rare case many parameters (fields of an HTML form, for example) would have the same name.
- **HttpSession getSession(boolean)**: returns the current session. Creates one if none exist and the parameter is true.
- **get/setAttribute(...)**: Managers the request scoped attributes.
- **BufferedReader getReader()**: Retrieves the body of the request as character steam.
- **RequestDispatcher getRequestDispatcher(String pathname)**: Returns a RequestDispatcher object that acts as a wrapper for the resource located at the given path. A RequestDispatcher object can be used to forward a request to the resource or to include the resource in a response. The resource can be dynamic or static. The pathname specified may be relative, although it cannot extend outside the current servlet context. If the path begins with a "/" it is interpreted as relative to the current context root. This method returns null if the servlet container cannot return a RequestDispatcher. This is essentially the same method as ServletContext.getRequestDispatcher().

5.1.7. The HttpServletResponse interface

HttpServletResponse (extends ServletResponse) represents the dynamically generated response. This usually is a HTML page that is sent back to the client.

It is also used for error pages and redirection. In addition to an HTML page, a response object may also be an HTTP error response or a redirection. In case of a redirection, the new URL can point to another servlet or Java Server Page (JSP).

- **void setContentType("text/html")**: sets the content type of the response being sent to the client. (Must always be done before getting the writer, and is also done in compiled JSPs)
- **PrintWriter getWriter()**: returns a PrintWriter object that can send character text to the client.
- **void setHeader(java.lang.String name, java.lang.String value)**: Sets a response header with the given name and value. If the header had already been set, the new value overwrites the previous one.
- **String encodeURL(String)**: This method is important in its relationship to session tracking. This is described in a different chapter.
- **HttpServletResponse.sendRedirect(String)**: Sends a temporary redirect response to the client using the specified redirect location URL. This method can accept relative URLs; the servlet container will convert the relative URL to an absolute URL before sending the response to the client.

5.1.8. The RequestDispatcher interface

The RequestDispatcher dispatches an existing request to some other servlet.

- **void forward(HttpServletRequest, HttpServletResponse)**: forwards the request and the response to another servlet or JSP. The response to the client will come exclusively from the second servlet or JSP. The forward() method should be called before the response has been committed to the client, meaning before response body output has been flushed. If the response already has been committed, this method throws an IllegalStateException. Uncommitted output in the response buffer is automatically cleared before the forward.
- **void include(HttpServletRequest, HttpServletResponse)**: includes the result of another servlet or JSP in the result of the first servlet. The included servlet cannot change the response status code or set headers; any attempt to make a change is ignored.

An object implementing RequestDispatcher can be retreived from two sources with a subtle difference:

- **ServletRequest.getRequestDispatcher()**: the pathname specified may be relative, although it cannot extend outside the current servlet context. If the path begins with a "/" it is interpreted as relative to the current context root. Typically used to forward to another servlet or JSP of the same application.
- **ServletContext.getRequestDispatcher()**: The pathname must begin with a / and is interpreted as relative to the current context root. The request details are not passed.

5.2. Lifecycle and Events

- Before a servlet can be loaded, the servlet engine must first locate its class. This class may reside on the local filesystem, a remote filesystem or be some other network resource.
- Once loaded, the servlet engine instantiates an instance of that servlet class.
- Once a servlet has been initialised it is ready to handle client requests. Servlet initialisation means that servlet instance is running with the servlet engine and is ready to accept requests.
- A servlet instance persists across client requests serviced in a multi-threaded manner within the servlet engine.

5.3. Securing an Application

5.4. Errors and Logging

5.5. Web fragments

5.5.1. Concept behind the web fragments

Servlets 3.0 specification introduces a new way to make your web applications more modular. Prior to Servlets 3.0 application developer had to manually edit *web.xml* deployment descriptor in order to add support for particular web framework framework to his/her web application.

For example the following code had to be manually added to the *web.xml* in order to enable Java Server Faces support.

Source

```
<web-app>
   <servlet>
       <servlet-name>JSF Servlet</servlet-name>
       <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
       <load-on-startup>1</load-on-startup>
   </servlet>

   <servlet-mapping>
       <servlet-name>JSF Servlet</servlet-name>
       <url-pattern>/myApp/*</url-pattern>
   </servlet-mapping>
</web-app>
```

Such approach has been violating the modularity of the application.

5.5.2. Using web fragments

Web fragment is a part of the *web.xml* file that can be created in the separate XML unit. Consider the example of web fragment below.

Source

```
<web-fragment>
   <servlet>
       <servlet-name>Foo Framework Servlet</servlet-name>
       <servlet-class>com.foo.FooServlet</servlet-class>
   </servlet>

   <listener>
       <listener-class>com.foo.FooServletListener</listener-class>
   </listener>
</web-fragment>
```

The fragment file should be named web-fragment.xml and added to the META-INF directory of the framework Jar file. It is then the reponsibility of the container to scan the classpath and enable the web fragments.

5.6. Architecture

5.6.1. CGI

When an HTTP requests arrives on the server, it is transported in a TCP/IP packet. It reaches the target TCP port. The standard for HTTP is the port 80. During this course you usually have worked with the port 8080. A process is listening on this port: the web server. In our case it is Tomcat.

What if the web server does not want to process the request, but would like to delegate the work to an external module? CGI (Common Gateway Interface) is a standard method to delegate such work to an external module.

Historically, a web server's main role is to send static files from the file system. From time to time, when a program's execution is required, a request is sent to a special url. A common convention is to include *cgi-bin* in the path, or end the request with ".cgi". When the web server figures out that the request should be processed by a CGI program, it calls that program (using the standard CGI conventions). Each call starts a new process, because the web server's process should not be blocked and continue to serve other requests. Such on OS process is heavyweight to start/stop on the server and performance consuming.

Historically, Java web containers (as Tomcat) were presented as a big improvement over CGI. The web server (Tomcat) starts a new thread (lightweight) for each request.

5.6.2. AJAX

How much should be processed on the client (browser) side, and with what technology?

The initial idea of Java inventors was to execute Java inside the browser, through Applets.

Users/developers did not like Applets at all and very soon (end of nineties), it was clear that most of the processing would be made on the server. The client "only" had to display the HTML generated by the server.

Soon (2003), users wanted more reactive pages, where the whole page does not have to be replaced/reloaded for every interraction. JavaScript was used to process some UI logic in the browser and apply changes to the HTML. This is named AJAX (Asynchronous JavaScript and XML) But JavaScript (which is not Java at all) and browser's APIs were not standard at all on different browsers (Internet Explorer, Firefox, etc.). To ease the pain of writing JavaScript code, libraries such as jQuery have been created.

Nowadays, some web applications are more client-side than server-side. There is so much JavaScript code to write, that Google has developped a compiler taking Java code as input and producing JavaScript code, to shield developers from the JavaScript hell. This is GWT (Google Web Toolkit).

5.6.3. Multi-Tiers

A multi-tier system separates areas of logic. In the Java EE environment, logic is often separated into tiers for client, presentation, business, and data. Typically, Java EE multi-tiered applications are distributed over three locations: client machines, the Java EE server, and a database machine on the back-end. This is a 3 tiers architecture:

We could add an EJB container in the Java EE server to run the business logic. The presentation logic code would remain in the web container. That architecture has 4 tiers. The web and EJB containers would be located in the same machine or not.

5.7. Packaging

Need to write topic about war files, manuyal deployment into Tomcat (not through eclipse) and the structure of the web.xml file.

6. Bonus

See video in the French version.