Security Lab Series

Introduction to Web Technologies

Prof. Lixin Tao
Pace University
http://csis.pace.edu/lixin

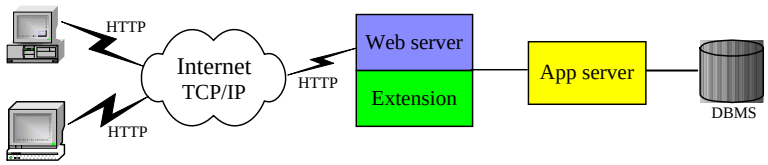
Contents

1 Concepts

Web servers and web browsers are communicating client-server computer programs for distributing documents and information, generally called web data, over the Internet. Web data are marked up in the HTML language for presentation and interaction with people in web browsers. Each web server uses an IP address or domain name as well as a port number for its identification. People use web browsers to send data requests to web servers with the HTTP protocol, and the web servers running on server computers either retrieve the requested data from local disks or generate the data on-the-fly, mark up the data in HTML, and send the resulting HTML files back to the web browsers to render. *Apache, Tomcat* and *IIS* are popular web server programs, and *IE* and *Firefox* are popular web browsers.

1.1 Web Architecture

A typical web application involves four tiers as depicted in the following web architecture figure: web browsers on the client side for rendering data presentation coded in HTML, a web server program that generates data presentation, an application server program that computes business logic, and a database server program that provides data persistency. The three types of server programs may run on the same or different server machines.



Tier 1 Web browser | Tier 2 Web server | Tier 3 App Server | Tier 4 Database

Web browsers can run on most operating systems with limited hardware or software requirement. They are the graphic user interface for the clients to interact with web applications. The basic functions of a web browser include:

- Interpret HTML markup and present documents visually;
- Support hyperlinks in HTML documents so the clicking on such a hyperlink can lead to the corresponding HTML file being downloaded from the same or another web server and presented;
- Use HTML form and the HTTP protocol to send requests and data to web applications and download HTML documents;
- Maintain cookies (name value pairs, explained later) deposited on client computers by a web application and send all cookies back to a web site if they are deposited by the web application at that web site (cookies will be further discussed later in this chapter);
- Use plug-in applications to support extra functions like playing audio-video files and running Java applets;
- Implement a *web browser sandbox* security policy: any software component (applets, JavaScripts, ActiveX, …) running inside a web browser normally cannot access local clients' resources like files or keyboards, and can only communicate directly with applications on the web server from where it is downloaded.

The web server is mainly for receiving document requests and data submission from web browsers through the HTTP protocol on top of the Internet's TCP/IP layer. The main function of the web server is to feed HTML files to the web browsers. If the client is requesting a static existing file, it will be retrieved on a server hard disk and sent back to the web browser right away. If the client needs customized HTML pages like the client's bank statement, a software component, like a JSP page or a servlet class (the "Extension" box in the web architecture figure), needs to retrieve the client's data from the database and compose a response HTML file on-the-fly.

The application server is responsible for computing the business logics of the web application, like carrying out a bank account fund transfer and computing the shortest route to drive from one city to another. If the business logic is simple or the web application is only used by a small group of clients, the application server is usually missing and business logics are computed in the web server extensions (PHP,

JSP or servlet, …). But for a popular web application that generates significant computation load for serving each client, the application server will take advantage of a separate hardware server machine to run business logics more efficiently. This is a good application of the divide-and-conquer problem-solving methodology.

Question 1: What are the four tiers of the web architecture?

Question 2: List programs for each of the four web tiers that our *ubuntu10* VM has installed.

Question 3: What is the difference between a web server and an application server?

Question 4: What is the main function of HTML?

Question 5: What is the main function of HTTP?

Question 6: Why many small companies only use web servers and don't use application servers?

Question 7: Is HTTP a transportation layer protocol or an application protocol?

Question 8: Does JavaScript run in web browsers or on web servers?

Question 9: Can JavaScript access the web browser user's file system?

Question 10: Can JavaScript communicate with the web site where it is downloaded?

Question 11: Can JavaScript communicate with the web site if the JavaScript is not downloaded from the web site?

Question 12: What is the web browser sandbox?

Question 13: Web browser or web server, who generated cookies?

Question 14: When and how are cookies sent to a web server?

Question 15: Can a cookie downloaded from web site A be sent to web site B by a web browser?

1.2 Uniform Resource Locators (URL)

A web server program runs multiple web applications (sites) hosted in different folders under the web server program's document root folder. A server computer may run multiple server programs including web servers. Each server program on a server computer uses a port number, between 0 and 65535, unique on the server machine as its local identification (by default a web server uses port 80). Each server computer has an IP address, like 198.105.44.27, as its unique identifier on the Internet. Domain names, like www.pace.edu, are used as user-friendly identifications of server computers, and they are mapped to IP addresses by a Domain Name Server (DNS). A Uniform Resource Locator (URL) is an address for uniquely identifying a web resource (like a web page or a Java object) on the Internet, and it has the following general format:

http://domain-name:port/application/resource?query-string

where *http* is the protocol for accessing the resource (*https* and *ftp* are popular alternative protocols standing for *secure HTTP* and *File Transfer Protocol*); *application* is a server-side folder containing all resources related to a web application; *resource* could be the name (alias or nickname) of an HTML or script/program file residing on a server hard disk; and the optional query string passes user data to the web server. An example URL is http://www.amazon.com/computer/sale?model=dell610.

There is a special domain name "`localhost`" that is normally defined as an alias of local IP address `127.0.0.1`. Domain name "`localhost`" and IP address `127.0.0.1` are for addressing a local computer, very useful for testing web applications where the web browser and the web server are running on the same computer.

Most computers are on the Internet as well as on a local area network (LAN), like home wireless network, and they have an external IP address and a local IP address. To find out what is your computer's external IP address on the Internet, use a web browser to visit http://whatismyip.com. To find out what is your local (home) IP address, on Windows, run "ipconfig" in a DOS window; and on Linux, run "sudo ifconfig" in a terminal window.

Question 16: What is the general structure of an URL?

Question 17: Why we need port numbers in networking?

Question 18: What is the default port number of a web server?

Question 19: How is a domain name mapped to an IP address?

Question 20: Could a server computer have multiple IP addresses?

Question 21: What is the function of domain name *localhost*?

Question 22: Domain name *localhost* is mapped to which IP address?

Question 23: How to find your computer's IP address on the Internet?

Question 24: How to find your Linux computer's IP address in your home wireless network?

Question 25: How to find your Windows computer's IP address in your home wireless network?

1.3 HTML Basics

HTML is a markup language. An HTML document is basically a text document marked up with instructions as to document logical structure and document presentation. The following is the contents of file "~/tomcat/webapps/demo/echoPost.html" in the *ubuntu10* VM.

```
<html>
<head>
<body>
  <form method="post" action="http://localhost:8080/demo/echo">
    Enter your name: <input type="text" name="user"/> <br/><br/>
```

```
      <input  type="submit"  value="Submit"/>
      <input  type="reset"   value="Reset"/>
    </form>
  </body>
  </html>
```

An HTML *tag name* is a predefined keyword, like `html`, `body`, `head`, `title`, `p`, and `b`, all in lower-case. A tag name is used in the form of a *start tag* or an *end tag*. A start tag is a tag name enclosed in angle brackets < and >, like `<html>` and `<p>`. An end tag is the same as the corresponding start tag except it has a forward slash / immediately before the tag name, like `</html>` and `</p>`.

An *element* consists of a start tag and a matching end tag based on the same tag name, with optional text or other elements, called *element value*, in between them. The following are some element examples:

```
  <p>This is free text</p>
```

```
  <p>This element has a nested <b>element</b></p>
```

While the elements can be nested, they cannot be partially nested: the end tag of an element must come after the end tags of all of its nested elements (*first starting last ending*). The following example is not a valid element because it violates the above rule:

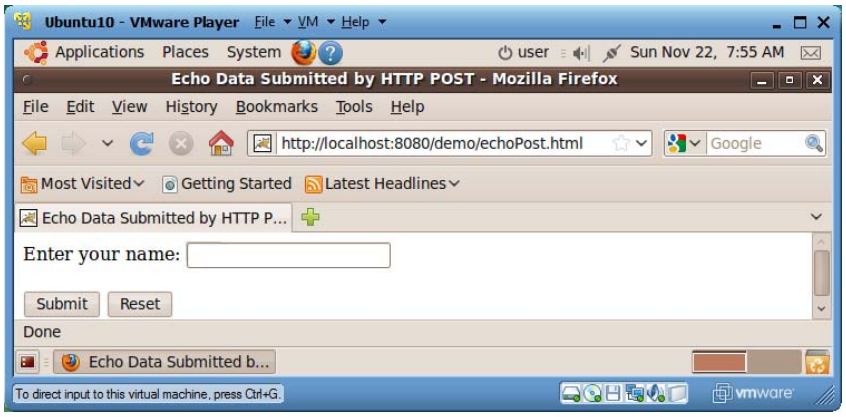
```
  <p>This is not a valid <bold>element<p><bold>
```

The *newline* character, the *tab* character and the *space* character are collectively called the *white-space characters*. A sequence of white-space characters acts like a single space for web browser's data presentation. Therefore, in normal situations, HTML document's formatting is not important (it will not change its presentation in web browsers) as long as you don't remove all white-space characters between successive words.

If an element contains no value, the start tag and the end tag can be combined into a single one as `<tagName/>`. As an example, we use `<br/>` to insert a line break in HTML documents.

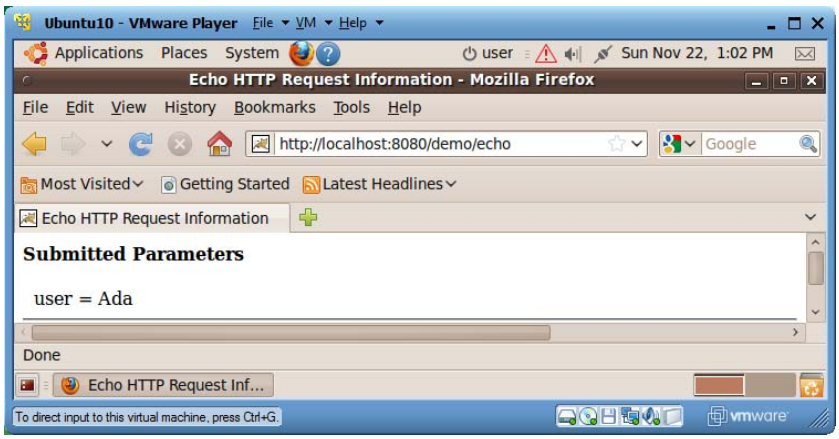
The start tag of an element may contain one or more *attributes*, each in the form "`attributeName="attributeValue""`. The above form element has two attributes: `method` and `action`.

An HTML document must contain exactly one top-level `html` element, which in turn contains exactly one `body` element. Most of the other contents are nested in the `body` element. If you load the above file "echoPost.html" in a web browser you will see the following:



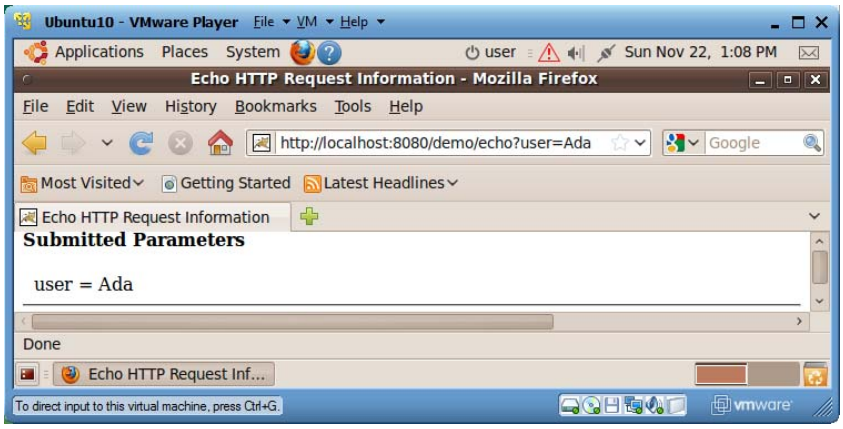
The `form` element is the most important mechanism for interaction between people and web applications. A form typically contains a few `input` elements and at least one submit button. A `form` element usually has two attributes: the `method` attribute for specifying HTTP method for submitting the form data to the web application (only values normally used are "get" and "post"); and the `action` attribute for specifying the form data submission destination, or the URL of a web application. In this example, when people click on the submit button, the form data will be sent to resource "echo" of the same web application "demo" deployed on your Ubuntu VM's Tomcat web server, which will echo back all information the web browser sent to the web server. If the `action` value doesn't specify the domain name/IP address or the web application, then the web application from where this HTML file came from will receive the form data. The first `input` element of type "text" has been rendered as a text field, the second `input` element of type "submit" has been rendered as a submit button, and the third `input` element of type "reset" has been rendered as a reset button. The `value` attribute of the `input` elements determines what string will be displayed on the element's image. The `name` attribute of the `input` element specifies the variable name with which web server programs can access what people type/enter in the element. When the submit button is clicked, the form data will be packaged as an HTTP request and sent to the web resource specified by the `action` attribute with the method specified by the `method` attribute.

If you type "Ada" in the name filed and click on the submit button, you will receive the HTTP response partially displayed below.



If you load file "echoGet.html" from the same web application folder "demo", the HTML file contents is basically the same except the method attribute for the form is changed from "post" to "get". If you enter "Ada" in the name field and click on the submit button again, you will notice that the query string

"`?user=Ada`" has been appended to the end of the URL. This is a major difference from HTTP POST method, and you will learn more about HTTP GET/POST soon.



An HTML file can contain hyperlinks to other web pages so users can click on them to visit different web pages. A hyperlink has the general structure of `<a href="url">Hyperlink Text</a>`. The following is an example hyperlink. Since its `href` value is not a web page, the *welcome page* of the Google web site, which is the default page sent back if a browser visits the web site without specifying a specific interested page, will be sent back to the web browser.

```
<a href="http://www.google.com">Google</a>
```

When you click on a hyperlink, an HTTP GET request will be sent to the web server with all values to be submitted in the form of query strings.

Question 26: What is the difference between HTML tag name and element?

Question 27: HTML attributes are specified in start tag or end tag?

Question 28: What are the two main HTML mechanisms for supporting interactions between a web browser and a web server?

Question 29: Can a web server initiate a communication with a web browser?

Question 30: What are the two main attributes of HTML's form element and what are their functions?

Question 31: How to create a text field for users to enter a value?

Question 32: How to create a submit button for users to submit the values in an HTML form?

Question 33: What is the meaning of the *name* attribute of HTML input elements?

Question 34: How to create an HTML hyperlink?

Question 35: When you click on a hyperlink, do you generate an HTTP POST or HTTP GET request?

1.4 HTTP Protocol

Web browsers interact with web servers with a simple application-level protocol called HTTP (HyperText Transfer Protocol), which runs on top of TCP/IP network connections. When people click on the submit button of an HTML form or a hyperlink in a web browser, a TCP/IP virtual communication channel is created from the browser to the web server specified in the URL; an HTTP GET or POST request is sent through this channel to the destination web application, which retrieves data submitted by the browser user and composes an HTML file; the HTML file is sent back to the web browser as an HTTP response through the same TCP/IP channel; and then the TCP/IP channel is shut down.

The following is the HTTP POST request sent when you type "Ada" in the text field and click on the submit button of the previous file "echoPost.html".

```
POST   /demo/echo   HTTP/1.1
Accept:  text/html
Accept:  audio/x
User-agent:  Mozilla/5.0
Referer:  http://localhost:8080/demo/echoPost.html
Content-length:  8

user=Ada
```

The first line, the request line, of a HTTP request is used to specify the submission type, GET or POST; the specific web resource on the web server for receiving and processing the submitted data; and the latest HTTP version that the web browser supports. As of 2010, version 1.1 is the latest HTTP specification. The following lines, up to before the blank line, are HTTP *header lines* for declaring web browser capabilities and extra information for this submission, each of form "name: value". The first two `Accept` headers declare that the web browser can process HTML files and any standard audio file formats from the web server. The `User-agent` header declares the software architecture of the web browser. The `Referer` (yes this misspelled word is used by the HTTP standard) header specifies the URL of a web page from which this HTTP request is generated (this is how online companies like Amazon and Yahoo collect money for advertisements on their web pages from their sponsors). Any text after the blank line below the header lines is called the *entity body* of the HTTP request, which contains user data submitted through HTTP POST. The `Content-length` header specifies the exact number of bytes that the entity body contains. If the data is submitted through HTTP GET, the entity body will be empty and the data go to the query string of the submitting URL, as you saw earlier.

In response to this HTTP POST request, the web server will forward the submitted data to resource `echo` of web application `demo`, and the resource `echo` (a Java servlet) will generate dynamically an HTML page for most data it can get from the submission and let the web server send the HTML page back to the web browser as the entity body of the following HTTP response.

```
HTTP/1.1   200   OK
Server: NCSA/1.3
Mime_version: 1.0
Content_type: text/html
Content_length: 2000

<HTML>
……
</HTML>
```

The first line, the response line, of an HTTP response specifies the latest HTTP version that the web server supports. The first line also provides a web server processing status code, the popular values of which include 200 for OK, 400 if the server doesn't understand the request, 404 if the server cannot find the requested page, and 500 for server internal error. The third entry on the first line is a brief message explaining the status code. The first two header lines declare the web server capabilities and meta-data for the returned data. In this example, the web server is based on a software architecture named "NCSA/1.3", and it supports *Multipurpose Internet Mail Extension* (MIME) specification v1.0 for web browsers to submit text or binary data with multi-parts. The last two header lines declare that the entity body contains HTML data with exactly 2000 bytes. The web browser will parse this HTTP response and present the response data.

The HTTP protocol doesn't have memory: the successive HTTP requests don't share data.

HTTP GET was initially designed for downloading static web pages from web servers, and it mainly used short query strings to specify the web page search criteria. HTTP POST was initially designed for submitting data to web servers, so it used the request entity body to send data to the web servers as a data stream, and its response normally depended on the submitted data and the submission status. While both HTTP GET and HTTP POST can send user requests to web servers and retrieve HTML pages from web servers for a web browser to present, they have the following subtle but important differences:

- HTTP GET sends data as query strings so people can read the submitted data over submitter's shoulders.
- Web servers have limited buffer size, typically 512 bytes, for accommodating query string data. If a user submits more data than that limit, either the data would be truncated, or the web server would crash, or the submitted data could potentially overwrite some computer code on the server and the server was led to run some hideous code hidden as part of the query string data. The last case is the so-called *buffer overflow*, a common way for hackers to take over the control of a server and spread virus or worms.
- By default web browsers keep (cache) a copy of the web page returned by an HTTP GET request so the future requests to the same URL can be avoided and the cached copy could be easily reused. While this can definitely improve the performance if the requested web page doesn't change, it could be disastrous if the web page or data change with time.

Question 36: When you click on a submit button of an HTML form, you find your form data appears in the web browser's URL address field. Is your form using method Get or POST?

Question 37: You use an HTML form to check the value of a specific stock. You found the stock price not changing for extended period of time even though you saw on TV that the stock's price had changed. What could be the problem?

Question 38: What are the main differences between HTTP GET and HTTP POST?

Question 39: HTTP GET and HTTP POST, which is more secure?

Question 40: Which HTTP request method can be used to launch buffer overflow attack to a web server?

Question 41: What is an HTTP request's *entity body* for?

Question 42: What are an HTTP request's header lines for?

Question 43: How can a web browser or a web server know its communication partners capabilities regarding HTTP version and data type support?

Question 44: What is the function of HTTP response header line for "referee"?

1.5 Session Data Management

Most web applications need a user to interact with it multiple times to complete a business transaction. For example, when you shop at Amazon, you choose one book at a time by clicking on some HTML form's submission buttons/hyperlinks in a web browser, and Amazon will process your submitted data and send you another HTML form for further shopping. A sequence of related HTTP requests between a web browser and a web application for accomplishing a single business transaction is called a *session*. All data specified by the user is called the *session data*. Session data are private so they must be protected from other users. A session normally starts when you first visit a web site in a particular day, and terminates when you pay off your purchase or shut down your web browser. Since the HTTP protocol has no memory, web applications have to use some special mechanisms to securely maintain the user session data.

1.5.1 Cookies

A *cookie* is a pair of name and value, as in (name, value). A web application can generate multiple cookies, set their life spans in terms of how many milliseconds each of them should be alive, and send them back to a web browser as part of an HTTP response. If cookies are allowed, a web browser will save all cookies on its hosting computer, along with their originating URLs and life spans. When an HTTP request is sent from a web browser of the same type on the same computer to a web site, all live cookies originated from that web site will be sent to the web site as part of the HTTP request. Therefore session data can be stored in cookies. This is the simplest approach to maintain session data. Since the web server doesn't need to commit any resources for the session data, this is the most scalable approach to support session data of large number of users. But it is not secure or efficient for cookies to go between a web browser and a web site for every HTTP request, and hackers could eavesdrop for the session data along the Internet path.

1.5.2 Hidden Fields

Some web users have great concern of the cookie's security implications and they disable cookie support on their web browsers. A web application can check the header fields of HTTP requests to detect whether cookies are supported by the requesting web browser. If the cookies are disabled, the web application will normally use form hidden fields to store session data. Upon receiving submitted data through an HTTP request, the web application will generate a new HTML form for the user to continue the business transaction, and it will populate all useful session data in the new HTML form as hidden fields (`input` elements of type "hidden"). When the user submits the form again, all the data that the user just entered the form, as well as all data saved in the form as hidden fields, will be sent back to the web application again. Therefore this hidden fields approach for maintaining session data shares most of the advantages and disadvantages of the cookie approach.

1.5.3 Query Strings

Sometimes query strings can also be used to maintain small amount of session data. This is particular true for maintaining the short session IDs that will be introduced below. But since most business transactions are implemented with HTML forms, this approach is less useful.

1.5.4 Server-Side Session Objects

For improving the security of session data and avoiding the wasted network bandwidth for session data to move back and forth between a web browser and a web server, you can also save much of the session data on the web server as server-side *session objects*. A session object has a unique session ID for identifying a specific user. A session object is normally implemented as a hash table (lookup table) consisting of (name, value) pairs. A single cookie, hidden field of a form, or query string of a hyperlink can be used to maintain the session ID. Since session ID is a fixed size small piece of data, it will not cause much network overhead for going between a web browser and a web server for each HTTP request. For securing the session data, you need to make sure that the session ID is unique and properly protected on the client site. Since this approach stores all session data on the web server, it takes the most server resources and is relatively harder to serve large number of clients concurrently.

Question 45: What is the meaning of a web session?

Question 46: What is web session data?

Question 47: When you shop at Amazon, how does Amazon maintain your session data (products that you have added to your virtual shopping cart)?

Question 48: What are the main mechanisms for supporting session data management?

Question 49: What is web session ID and why its security is important?

Question 50: Which session data management mechanism is relatively more secure?

Question 51: Which session management mechanism is relatively more scalable (supporting huge number of clients' session data without committing proportional amount of resources on the web server)?

Question 52: The life span of a cookie is determined by a web server, a web application, or a web browser?

Question 53: What are the pros and cons when you decide on the life span of a cookie?

2 Lab Objectives

In this lab you will

1. Compare HTTP GET and HTTP POST requests;
2. Observe HTTP communications with proxy server *Paros*;
3. Experiment with cookies through web applications;

4. Compare web browser and web server interactions with HTML forms and with hyperlinks;
5. Learn how to use JavaScript to validate form data in the web browsers;
6. Learn how to create a static web site;
7. Learn how to create your first JSP web application on Tomcat;
8. Learn how to create your first servlet web application on Tomcat.

3 Lab Setup

You will use the *ubuntu10* VM and there is no extra lab setup for this set of lab exercises.

4 Lab Guide

4.1 Comparing HTTP GET and HTTP POST Requests

1. Launch the *ubuntu10* VM with username "user" and password 12345678.
2. Start web browser and visit "http://localhost:8080/demo/echoPost.html".
3. Use the *gedit* (menu item "Applications|Accessories|gedit Text Editor") text editor to open and review the contents of file "~/tomcat/webapps/demo/echoPost.html".
4. Enter your name in the name text field, and click on the submit button. Observe that the URL doesn't include your name. Read the returned web page for information that was submitted from your web browser to the Tomcat web server.
5. Redo steps 2-4 but visit http://localhost:8080/demo/echoGet.html ("~/tomcat/webapps/demo/echoGet.html").

Question 54: List all differences between HTTP Get and HTTP POST requests that you observe through this exercise.

Question 55: If you use HTTP GET to submit form data that contains a space, such as "John Jay", how the space is represented in query strings?

Question 56: If you change the `action` value of the form in "demo/echoGet.html" or "demo/echoPost.html" from "http://localhost:8080/demo/echo" to "/demo/echo", do you see any differences in the behavior of form data submission? (You need to use "sudo gedit echoGet.html" to change the file since the file was created by "root")

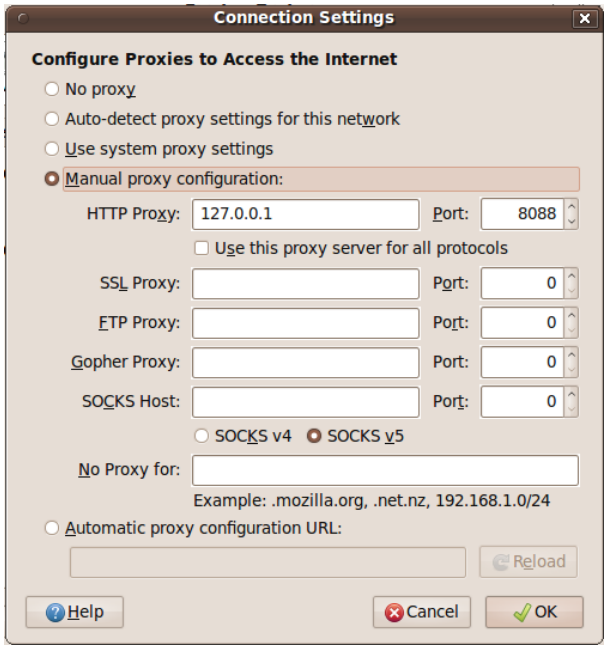
Question 57: If you change the `action` value of the form in "demo/echoGet.html" or "demo/echoPost.html" from "http://localhost:8080/demo/echo" to "/echo", do you see any differences in the behavior of form data submission?

Question 58: If you change the `action` value of the form in "demo/echoGet.html" or "demo/echoPost.html" from "http://localhost:8080/demo/echo" to "echo", do you see any differences in the behavior of form data submission?

4.2 Observing HTTP Communications with *Paros*

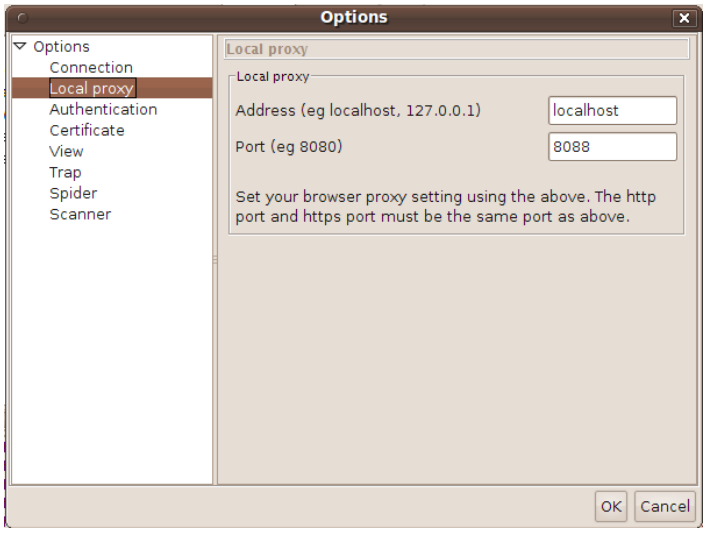
In this exercise, you will install the *Paros* proxy server on your *Ubuntu10* VM, and use *Paros* to observer HTTP communications. The *Paros* proxy server will run at port 8088. You will set up the *Firefox* web browser so that when you use the browser to visit any web site, the HTTP request will be first forwarded to the *Paros* proxy server running at port 8088, which displays the HTTP request information in its graphic user interface, lets the user to have a chance to review and modify the request, sends the request to its destination server, and forwards the HTTP response from the web server back to the *Firefox* web browser. In this exercise you mainly use *Paros* to intercept HTTP GET/POST requests.

1. In your *ubuntu10* VM, use a web browser to visit http://sourceforge.net/projects/paros/files/, and then click on file "paros-3.2.13-unix.zip" to download it to your VM's folder *Desktop* or *Downloads*.
2. In a file browser, right-click on the downloaded file and choose menu item "Extract Here" to create new folder "paros".
3. In a file browser, move folder "paros" to "~/tools/paros".
4. Now you need to edit your browser's proxy settings. Assume you are using *Firefox* V3.5.4 (VM ubuntu10). Launch your *Firefox* web browser, and follow its menu item path "Edit|Preferences|Advanced|Network Tab|Settings button" to reach the "Connection Settings" window. Check the "Manual proxy configuration" checkbox. In the HTTP Proxy" text field, enter **127.0.0.1**. In its "Port" text field, enter **8088**. Delete values in the "No proxy for" text field. Now your "Connection Settings" window looks like the following one



5. Click on the OK button to close the "Connection Settings" window. Click on the Close button to close the "Firefox Preferences" window.
6. Use menu item "Applications|Accessories|Terminal" to start a terminal window. Run command "cd ~/tools/paros" to change folder to "~/tools/paros". Run command "java -jar paros.jar" (or "sh startserver.sh"), accept the license agreement, and *Paros* will start to run with a graphic user interface. In the terminal window you should see error messages because by default *Paros* uses port 8080 which is already in use by the Tomcat web server. In the GUI of Paros, click on menu

item “Tools|Options…” to pop up the “Options” window, select “Local proxy” in the left pane, and enter 8088 in the right *Port* text field, as shown below.



7. Now click on the OK button to shutdown the Options window. Shut down *Paros*, and then restart it. Now *Paros* remembers its new port number and runs at port 8088. Please don’t shut down the terminal window that you used to launch *Paros*. Otherwise *Paros* will shut down too.
8. Test the *Paros* proxy server by visiting http://localhost with your *Firefox* web browser. Once the browser contacts your web server, *Paros* starts to display HTTP request information in its graphic user interface.
9. You have just enabled all HTTP traffic generated by Firefox to be sent to the running *Paros* proxy server which can analyze HTTP traffic before it is sent off to its final destination.
10. Reload http://localhost. Go back to *Paros*, click on *Sites* to reveal “http://localhost”, and select it.
11. Click on the Request tab. You will see the HTTP request sent from the browser to the local Apache web server that *Paros* reads while being transmitted.
12. Now redo exercise 4.1 with the configured web browser, and use *Paros* to observe the HTTP communications.

Important Note: After finishing this exercise, you should reset *Firefox* proxy setting so it stops using the proxy server. Otherwise you would not be able to visit web sites without running the proxy server at port 8088.

4.3 Working with Cookies

1. Launch the *ubuntu10* VM with username “user” and password 12345678.
2. Start web browser and visit “http://localhost:8080/demo”.
3. Use a *gedit* (menu item “Applications|Accessories|gedit Text Editor”) text editor to open and review the contents of file “~/tomcat/webapps/demo/index.html”.
4. Enter “name1” and “value1” in the name text field and value text field, and click on the submit button. Read the returned web page for information that was submitted from your web browser to Tomcat web server. You will see that the web server has not received cookies yet.
5. Use the browser back button to go back to the web page for “http://localhost:8080/demo”. Within 50 seconds. Enter “name2” and “value2” in the name text field and value text field, and click on

the submit button. Read the returned web page for information that was submitted from your web browser to Tomcat web server. You will see that the web browser has received cookie (name1, value1).

6. Use the browser back button to go back to the web page for "http://localhost:8080/demo". Wait for one minute or longer. Enter "name3" and "value3" in the name text field and value text field, and click on the submit button. Read the returned web page for information that was submitted from your web browser to Tomcat web server. You will see that the web server has received no cookies because the two cookies that you created have expired.
7. Use web browser to visit "http://localhost:8080/testCookie".
8. Use the *gedit* text editor to open and review the contents of file "~/tomcat/webapps/testCookie/index.html".
9. Enter your name in the name text field, and click on the submit button.
10. Within 10 seconds you click on the "Say Hello" button, and you will see a greeting to you.
11. Use the browser back button to go back to the previous web page. Wait for 15 seconds or longer for your name cookie to expire. Then click on the "Say Hello" button, and you will not be able to see the greeting to you.

Question 59: Can the web developer determine how long a cookie should be alive on the end users' computer?

Question 60: Is it a good idea to let your web applications' cookies live forever on end users' computers?

Question 61: Can you figure out how to delete all cookies live in your web browser?

Question 62: Can you figure out how to disable cookies for your web browser?

Question 63: Can you figure out where are your cookies saved by your web browser?

Question 64: If your computer has multiple types of web browsers installed, do they share cookies?

4.4 Submitting Data with HTML Form and Hyperlink

1. Launch the *ubuntu10* VM with username "user" and password 12345678.
2. Start web browser and visit "http://localhost:8080/tripler".
3. Use a *gedit* (menu item "Applications|Accessories|gedit Text Editor") text editor to open and review the contents of file "~/tomcat/webapps/tripler/index.html".
4. Enter "1" in the first text field, and click on the "Triple this integer" button to its left. Notice the query string "?number=1" appended to the end of the URL because an HTTP GET request was used.
5. Use the browser back button to go back to the web page for "http://localhost:8080/tripler". Enter "2" in the second text field, and click on the "Triple this integer" button to its left. Notice that no query string is used in the URL because it is an HTTP POST request.
6. Use the browser back button to go back to the web page for "http://localhost:8080/tripler". Click on the "Triple 4" hyperlink in the last line of the web page. Notice the query string "?number=4" appended to the end of the URL because HTTP GET request is always used when a hyperlink is clicked on.

Question 65: What type of HTTP request will be used when a user clicks on a hyperlink?

4.5 Validating Form Data with JavaScript

JavaScript is a simple scripting language that runs inside web browser security sandbox. It can be used to manipulate HTML file and validate form data before they are submitted to web applications. In this exercise you will learn how to use JavaScript to validate form data.

1. Launch the *ubuntu10* VM with username "user" and password 12345678.
2. In a file explorer, open "/home/user/tomcat/webapps". Right-click on any blank space in the explorer right pane and choose "Create Folder" to create new folder "/home/user/tomcat/webapps/test".
3. In the file explorer, open folder "~/tomcat/webapps/test". Right-click on any blank space in the explorer and choose "Create Document|Empty File" and create a new file with name "index.html".
4. Right-click on file "index.html" and choose menu item "Open With|gedit" to open file "index.html" in the gedit editor.
5. Type the following text into the file, and save the file. You can leave the editor open for later file modification.

```
<html>
<head><title>Echo Submitted Data</title></head>
<body>
  <form  method="get"  action="/demo/echo">
    Enter your name: <input   type="text"  name="user"/><br/><br/>
    <input  type="submit"  value="Submit"/>
    <input  type="reset"  value="Reset"/>
  </form>
</body>
</html>
```

6. Start web browser and visit "http://localhost:8080/test".
7. Type your name in the name text field and click on the submit button, and you will see your form data echoed back to you from the web server.
8. Use the browser's left arrow to go back to the web page of "http://localhost:8080/test".
9. Make sure the name text field is empty, and click on the submit button. You will see the incomplete form data is still submitted to the web server and no warning was given.
10. Now we are ready to add JavaScript code in the HTML file to make sure that the form would not submit unless the user has typed something in the name field. In the gedit editor, modify the file contents so it reads as below. Save the file. The JavaScript code is declared inside the *script* element which is nested in the *head* element. You don't need to pay attention to the *trim()* function which is used to remove the leading and trailing white space characters of a string (Java has this method, but JavaScript doesn't have it). First notice the newly added 3rd attribute of the *form* element: onsubmit="return dataCheck(this)". With this new attribute, when a submit button is clicked, the form (*this*) is first validated by JavaScript function *dataCheck()*. If the function returns true, the form data is valid and then actually submitted to the *action* target as an HTTP request. If the function returns false, the validation fails, the form is not submitted, and the user has the chance to revise the form data. In the body of function *dataCheck(form)*, parameter *form* represents the HTML form, and the data that the user has typed in the name text field is in variable "form.user" ("user" is the text field's name). The text field's value is first assigned into

a new variable *widget*. If the value is not empty, then the validation succeeds and the *dataCheck()* function returns true. If the value is an empty string, then the form has not been completed yet and thus is not valid. In this case the *alert()* function is used to pop up a warning window, the *focus()* method is called to position the cursor in the text field, the *select()* function is called to highlight the text field so the user can be ready to type value in the text field, and then the *dataCheck()* function returns false to declare validation failure.

```
<html>
<head><title>Echo Submitted Data</title>
<script language="Javascript" type="text/javascript">
  String.prototype.trim = function() {
    // remove string's leading spaces, then trailing spaces
    return this.replace(/^\s*/, "").replace(/\s*$/, "");
  }
  function dataCheck(form) {
    widget = form.user;
    if (widget.value.trim() == "") {
      alert("You must enter your name");
      widget.focus();  // position the cursor in the field
      widget.select(); // highlight the field
      return false;
    }
    return true;
  }
</script>
</head>
<body>
  <form method="post" action="/demo/echo"
        onsubmit="return dataCheck(this)">
    Enter your name: <input   type="text"  name="user"/><br/><br/>
    <input  type="submit"  value="Submit"/>
    <input  type="reset"  value="Reset"/>
  </form>
</body>
</html>
```

11. Use the web browser to visit http://localhost:8080/test again. Leave the name text field empty and click on the submit button. This time you will see the warning and the form is not submitted but let you revise.

Be warned that validating form data with JavaScript is not secure because people could save a copy of the HTML file on the hard disk, modify the file to remove the validation invocation, and then reload the HTML file. For sensitive form data the web applications must validate them on the web server.

Question 66: Which attribute of HTML form is used to implement JavaScript form data validation?

Question 67: Why JavaScript form data validation is not secure?

4.6 Creating Your First JavaServer Page Web Application

JavaServer Page (JSP) is a Java technology for generating HTML files on-the-fly based on data submitted by web browser users. In this exercise you will create an HTML form that submits data to a JSP page for presentation. A JSP page is basically an HTML file with small pieces of Java code pieces enclosed in <% and %> delimiters to compute dynamic values.

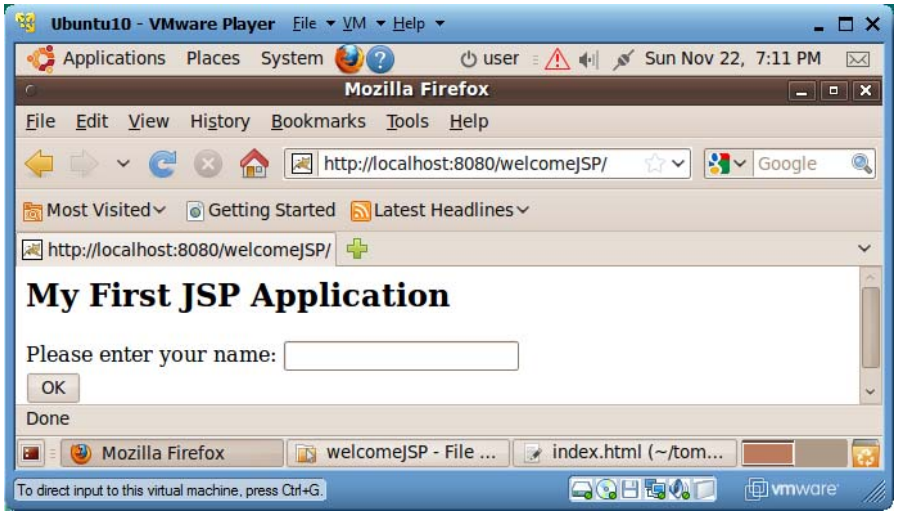
1. Launch the *ubuntu10* VM with username "user" and password 12345678.
2. In a file explorer, open "/home/user/tomcat/webapps". Right-click on any blank space in the explorer right pane and choose "Create Folder" to create new folder "/home/user/tomcat/webapps/welcomeJSP".
3. In the file explorer, open folder "~/tomcat/webapps/welcomeJSP". Right-click on any blank space in the explorer and choose "Create Document|Empty File" and create a new file with name "index.html".
4. Right-click on file "index.html" and choose menu item "Open With|gedit" to open file "index.html" in the gedit editor.
5. Type the following text into the file, and save the file.

```
<html>
<body>
<h2>My First JSP Application</h2>
<form method="post" action="welcome.jsp">
Please enter your name: <input type="text" name="name"/>
<br/>
<input type="submit" value="OK"/>
</form>
</body>
</html>
```

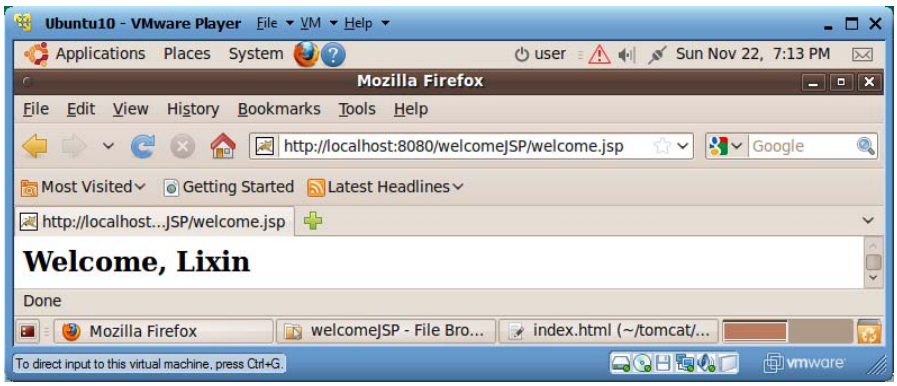
6. Repeat steps 3, 4 and 5 to create a new file "welcome.jsp" with the following contents. JSP *request* object represents all data submitted by a web browser. Method *request.getParameter("name")* retrieves the value that the user has entered the text field with name "name". JSP expression <%=*name*%> returns the current value of variable *name*.

```
<html>
<body>
<%
  String name = request.getParameter("name");
%>
<h2>Welcome, <%=name%></h2>
</body>
</html>
```

7. Use the web browser to visit http://localhost:8080/welcomeJSP and you will see the following window:



8. Type your name in the text field and click on the OK button, and you will see a window similar to the following one.



9. Congratulations and you have completed your first JSP web application!
10. Use the file explorer to visit folder "/home/user/tomcat/work/Catalina/localhost/welcomeJSP/org/apache/jsp". You will find two files "welcome_jsp.java" and "welcome_jsp.class". When the JSP page "welcome.jsp" is visited for the first time, the file is converted into a Java servlet source file "welcome_jsp.java" which is then compiled into binary file "welcome_jsp.class" for execution. This transformation will only happen for the first visit of the JSP page. The following is part of contents of file "welcome_jsp.java":

```
        out.write("<html>\n");
        out.write("<body>\n");
        String name = request.getParameter("name");
        out.write("\n");
        out.write("<h2>Welcome, ");
        out.print(name);
        out.write("</h2>\n
        out.write("</body>\n");
        out.write("</html>\n");
```

11. Therefore JSP is built on top of Java servlets and JSP just makes Java servlets easier to use by web developers.

Question 68: A JSP page is more like an HTML file or a Java source code file?

Question 69: In JSP, how to retrieve the value that the web browser user has typed in a text field?

Question 70: In JSP how to insert the value of a variable in the HTML file?

Question 71: Is JSP a technology independent of the servlet technology?

Question 72: When are JSP files converted into Java servlet files?

4.7 Creating Your First Servlet Web Application

From the last exercise we know servelts are the cornerstone of Java web technologies. In this exercise you will develop a servlet web application with the same function as the last *welcomeJSP* web application so you can better compare the two technologies.

1. Launch the *ubuntu10* VM with username "user" and password 12345678.
2. In a file explorer, open "/home/user/tomcat/webapps". Right-click on any blank space in the explorer right pane and choose "Create Folder" to create new folder "/home/user/tomcat/webapps/welcomeServlet".
3. In the file explorer, open folder "~/tomcat/webapps/welcomeServlet". Right-click on any blank space in the explorer and choose "Create Document|Empty File" and create a new file with name "main.html".
4. Right-click on file "main.html" and choose menu item "Open With|gedit" to open file "main.html" in the gedit editor.
5. Type the following text into the file, and save the file.

```
<html>
<body>
<h2>My First Servlet Application</h2>
<form method="post" action="welcome">
Please enter your name: <input type="text" name="name"/>
<br/>
<input type="submit" value="OK"/>
</form>
</body>
</html>
```

6. Create in folder "~/tomcat/webapps/welcomeServlet" a new folder "WEB-INF". All servlet based web applications have this folder for holding those files not directly accessible from web browsers.
7. Create in folder "~/tomcat/webapps/welcomeServlet/Web-INF" a new folder "classes". All Java classes to be run on the web server must be under this folder.
8. Create in folder "~/tomcat/webapps/welcomeServlet/Web-INF/classes" a new file "Welcome.java". Copy the following contents in this file and save the file. If a servlet needs to process HTTP POST requests, it needs to have a *doPost(request, response)* method. If a servlet

needs to process HTTP GET requests, it needs to have a *doGET(request, response)* method. We put all logics in the *doPost(request, response)* method and call this method inside the *doGet(request, response)* method to avoid redundant code. Both of the two methods have two parameters: *request* representing all data submitted through the HTTP request, including data in the HTTP request entity body and query string; and *response* representing the data to be sent back to the remote web browser through an HTTP response. Method *request.getParameter("name")* is first called to retrieve the value that the user has typed in the *name* text field. After setting the output data type and retrieving the output object of *response*, the remainder of the code just prints out an HTML file piece by piece. This section of code must remind you of the similar code we reviewed for the Java code produced from file "Welcome.jsp" in the last exercise.

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

public class Welcome extends HttpServlet {
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws IOException, ServletException {
    String name = request.getParameter("name");
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<body>");
    out.println("<h2>Welcome, " + name + "</h2>");
    out.println("</body>");
    out.println("</html>");
  }

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
    throws IOException, ServletException {
    doPost(request, response);
  }
}
```

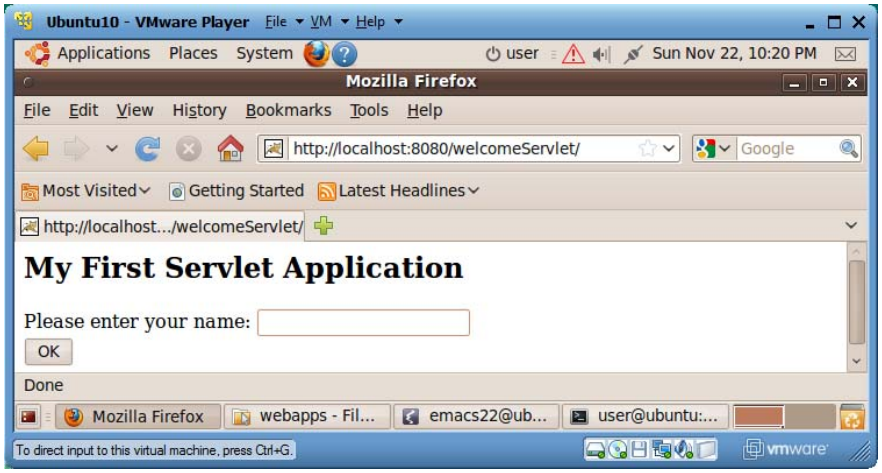
9. Create in folder "~/tomcat/webapps/welcomeServlet/Web-INF" a new file "web.xml" and copy the following contents into it. Each servlet web application needs this configuration file. You first declare that file "main.html" is the web application's welcome file: if a web browser visits this web application but not specifying which file to retrieve, the welcome file will be sent back by default. You then assign a name "welcome" to the servlet class "Welcome". In the last "servlet-mapping" element, you declare that if the URL of an HTTP request contains "/welcome", the request will be sent to the servlet named "welcome" for processing, which is "Welcome" in this case.

```
<web-app>
  <servlet>
    <servlet-name>welcome</servlet-name>
    <servlet-class>Welcome</servlet-class>
  </servlet>
```

```
    <servlet-mapping>
      <servlet-name>welcome</servlet-name>
      <url-pattern>/welcome</url-pattern>
    </servlet-mapping>

    <welcome-file-list>
      <welcome-file>main.html</welcome-file>
    </welcome-file-list>
  </web-app>
```

10. Now you need to compile the servlet into a bytecode file "Welcome.class". Use the file browser to open folder "~/tomcat/webapps/welcomeServlet/Web-INF/classes". Right-click on any blank space in the right pane, and choose menu item "Open in Terminal". A new terminal window will start with "~/tomcat/webapps/welcomeServlet/Web-INF/classes" as its working folder. Run command "javac Welcome.java" to compile Java source file "Welcome.java" into "Welcome.class".
11. You have completed your first servlet web application. Use a web browser to visit http://localhost:8080/welcomeServlet and you will see a web browser view similar to the following one. [Sometimes *Tomcat* could get confused when you are developing web applications. Restart the VM may resolve some of these problems.]



12. JSP and servlet web applications are normally deployed as Web Archive (WAR) files. To make a WAR file "welcomeServlet.war" for the current web application, start a terminal window in folder "~/tomcat/webapps/welcomeServlet, and run command "jar cvf welcomeServlet.war *". To deploy this web application in a different Tomcat web server, you only need to drop file "welcomeServlet.war" in that Tomcat installation's "webapps" folder, and this WAR file will be automatically extracted into web application folder "welcomeServlet" and the web application will start to work right away, assuming that Tomcat is running.
13. By now you have successfully completed your first servlet web application. Congratulations!

Question 73: What kind of files should be put under folder "WEB-INF"?

Question 74: Where should Java servlet files be located in a servlet web application?

Question 75: Appletes are Java classes to be downloaded to web browsers and run in web browser sandboxes. Should applet files be put under folder "WEB-INF"?

Question 76: What are the main methods that a Java servlet class should have?

Question 77: What are the functions of parameters `request` and `response` of servlet methods `doGet()` and `doPost()`?

Question 78: What is the main function of servlet methods `doGet()` and `doPost()`?

Question 79: What is the main function of configuration file "WEB-INF/web.xml"?

Question 80: What is a welcome file of a web application?

Question 81: What is URL pattern of a servlet?

Question 82: What is the relationship between a JSP page and a servlet class?

5 Review Questions

Question 83: In your *ubuntu10* VM, visit web application http://localhost:8080/bareJsp and study its source files. Explain the function and design of this web application.

Question 84: Develop a JSP web application that displays in a web browser an integer and a submit button. The integer is initially 0. Each time the user clicks on the button, the integer increases by 1. [Hint: To convert string "12" to integer 12, you can use Java code

```
int v = 0;
try  {  v = Integer.parseInt("12"); }
catch (Exception  e) { v = 0; }
```

]

Question 85: Develop a servlet web application that displays in a web browser an integer and a submit button. The integer is initially 0. Each time the user clicks on the button, the integer increases by 1. [Hint: If you want a servlet to create the default welcome web page, use the servlet's "servlet-name" element value as the value of the "welcome-file" element in file "web.xml". To convert string "12" to integer 12, you can use Java code

```
int v = 0;
try  {  v = Integer.parseInt("12"); }
catch (Exception  e) { v = 0; }
```

]